CHAPTER 9

The Graph Data Model

A graph is, in a sense, nothing more than a binary relation. However, it has a powerful visualization as a set of points (called nodes) connected by lines (called edges) or by arrows (called arcs). In this regard, the graph is a generalization of the tree data model that we studied in Chapter 5. Like trees, graphs come in several forms: directed/undirected, and labeled/unlabeled.

Also like trees, graphs are useful in a wide spectrum of problems such as computing distances, finding circularities in relationships, and determining connectivities. We have already seen graphs used to represent the structure of programs in Chapter 2. Graphs were used in Chapter 7 to represent binary relations and to illustrate certain properties of relations, like commutativity. We shall see graphs used to represent automata in Chapter 10 and to represent electronic circuits in Chapter 13. Several other important applications of graphs are discussed in this chapter.

9.1 What This Chapter Is About

The main topics of this chapter are

- The definitions concerning directed and undirected graphs (Sections 9.2 and 9.10).
- The two principal data structures for representing graphs: adjacency lists and adjacency matrices (Section 9.3).
- An algorithm and data structure for finding the connected components of an undirected graph (Section 9.4).
- A technique for finding minimal spanning trees (Section 9.5).
- A useful technique for exploring graphs, called "depth-first search" (Section 9.6).

- Applications of depth-first search to test whether a directed graph has a cycle, to find a topological order for acyclic graphs, and to determine whether there is a path from one node to another (Section 9.7).
- Dijkstra's algorithm for finding shortest paths (Section 9.8). This algorithm finds the minimum distance from one "source" node to every node.
- Floyd's algorithm for finding the minimum distance between any two nodes (Section 9.9).

Many of the algorithms in this chapter are examples of useful techniques that are much more efficient than the obvious way of solving the given problem.

9.2 Basic Concepts

Directed graph

A *directed graph*, consists of

1. A set N of *nodes* and

Nodes and arcs

2. A binary relation A on N. We call A the set of *arcs* of the directed graph. Arcs are thus pairs of nodes.

Graphs are drawn as suggested in Fig. 9.1. Each node is represented by a circle, with the name of the node inside. We shall usually name the nodes by integers starting at 0, or we shall use an equivalent enumeration. In Fig. 9.1, the set of nodes is $N = \{0, 1, 2, 3, 4\}$.

Each arc (u, v) in A is represented by an arrow from u to v. In Fig. 9.1, the set of arcs is

$$A = \{(0,0),\ (0,1),\ (0,2),\ (1,3),\ (2,0),\ (2,1),\ (2,4),\ (3,2),\ (3,4),\ (4,1)\}$$

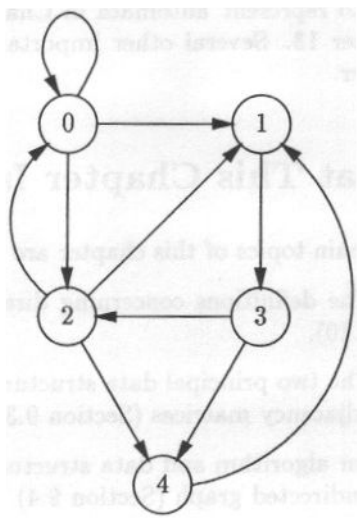


Fig. 9.1. Example of a directed graph.

Head and tail

In text, it is customary to represent an arc (u, v) as $u \rightarrow v$. We call v the *head* of the arc and u the *tail* to conform with the notion that v is at the head of the

arrow and u is at its tail. For example, $0 \rightarrow 1$ is an arc of Fig. 9.1; its head is node 1 and its tail is node 0. Another arc is $0 \rightarrow 0$; such an arc from a node to itself is called a *loop*. For this arc, both the head and the tail are node 0.

Loop

Predecessors and Successors

When $u \rightarrow v$ is an arc, we can also say that u is a *predecessor* of v, and that v is a *successor* of u. Thus, the arc $0 \rightarrow 1$ tells us that 0 is a predecessor of 1 and that 1 is a successor of 0. The arc $0 \rightarrow 0$ tells us that node 0 is both a predecessor and a successor of itself.

Labels

As for trees, it is permissible to attach a *label* to each node. Labels will be drawn near their node. Similarly, we can label arcs by placing the label near the middle of the arc. Any type can be used as a node label or an arc label. For instance, Fig. 9.2 shows a node named 1, with a label "dog," a node named 2, labeled "cat," and an arc $1 \rightarrow 2$ labeled "bites."

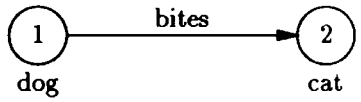


Fig. 9.2. A labeled graph with two nodes.

Again as with trees, we should not confuse the name of a node with its label. Node names must be unique in a graph, but two or more nodes can have the same label.

Paths

A *path* in a directed graph is a list of nodes $(v_1, v_2, \ldots, v_k)$ such that there is an arc from each node to the next, that is, $v_i \rightarrow v_{i+1}$ for $i = 1, 2, \ldots, k-1$. The *length* of the path is $k-1$, the number of arcs along the path. For example, $(0, 1, 3)$ is a path of length two in Fig. 9.1.

Length of a path

The trivial case $k = 1$ is permitted. That is, any node v by itself is a path of length zero from v to v. This path has no arcs.

Cyclic and Acyclic Graphs

A *cycle* in a directed graph is a path of length 1 or more that begins and ends at the same node. The *length of the cycle* is the length of the path. Note that a trivial path of length 0 is not a cycle, even though it "begins and ends at the same node." However, a path consisting of a single arc $v \rightarrow v$ is a cycle of length 1.

Length of a cycle

✦ **Example 9.1.** Consider the graph of Fig. 9.1. There is a cycle $(0, 0)$ of length 1 because of the loop $0 \rightarrow 0$. There is a cycle $(0, 2, 0)$ of length 2 because of the arcs $0 \rightarrow 2$ and $2 \rightarrow 0$. Similarly, $(1, 3, 2, 1)$ is a cycle of length 3, and $(1, 3, 2, 4, 1)$ is a cycle of length 4. ✦

Equivalent cycles

Note that a cycle can be written to start and end at any of its nodes. That is, the cycle $(v_1, v_2, \ldots, v_k, v_1)$ could also be written as $(v_2, \ldots, v_k, v_1, v_2)$ or as $(v_3, \ldots, v_k, v_1, v_2, v_3)$, and so on. For example, the cycle $(1, 3, 2, 4, 1)$ could also have been written as $(2, 4, 1, 3, 2)$.

Simple cycle

On every cycle, the first and last nodes are the same. We say that a cycle $(v_1, v_2, \ldots, v_k, v_1)$ is *simple* if no node appears more than once among $v_1, \ldots, v_k$; that is, the only repetition in a simple cycle occurs at the final node.

✦ **Example 9.2.** All the cycles in Example 9.1 are simple. In Fig. 9.1 the cycle $(0, 2, 0)$ is simple. However, there are cycles that are not simple, such as $(0, 2, 1, 3, 2, 0)$ in which node 2 appears twice. ✦

Given a nonsimple cycle containing node v, we can find a simple cycle containing v. To see why, write the cycle to begin and end at v, as in $(v, v_1, v_2, \ldots, v_k, v)$. If the cycle is not simple, then either

1. v appears three or more times, or
2. There is some node u other than v that appears twice; that is, the cycle must look like $(v, \ldots, u, \ldots, u, \ldots, v)$.

In case (1), we can remove everything up to, but not including, the next-to-last occurrence of v. The result is a shorter cycle from v to v. In case (2), we can remove the section from u to u, replacing it by a single occurrence of u, to get the cycle $(v, \ldots, u, \ldots, v)$. The result must still be a cycle in either case, because each arc of the result is present in the original cycle, and therefore is present in the graph.

It may be necessary to repeat this transformation several times before the cycle becomes simple. Since the cycle always gets shorter with each iteration, eventually we must arrive at a simple cycle. What we have just shown is that if there is a cycle in a graph, then there must be at least one simple cycle.

✦ **Example 9.3.** Given the cycle $(0, 2, 1, 3, 2, 0)$, we can remove the first 2 and the following 1, 3 to get the simple cycle $(0, 2, 0)$. In physical terms, we started with the cycle that begins at 0, goes to 2, then 1, then 3, back to 2, and finally back to 0. The first time we are at 2, we can pretend it is the second time, skip going to 1 and 3, and proceed right back to 0.

For another example, consider the nonsimple cycle $(0, 0, 0)$. As 0 appears three times, we remove the first 0, that is, everything up to but not including the next-to-last 0. Physically, we have replaced the path in which we went around the loop $0 \rightarrow 0$ twice by the path in which we go around once. ✦

Cyclic graph

If a graph has one or more cycles, we say the graph is *cyclic*. If there are no cycles, the graph is said to be *acyclic*. By what we just argued about simple cycles, a graph is cyclic if and only if it has a simple cycle, because if it has any cycles at all, it will have a simple cycle.

✦ **Example 9.4.** We mentioned in Section 3.8 that we could represent the calls

Calling graph

made by a collection of functions with a directed graph called the "calling graph." The nodes are the functions, and there is an arc $P \rightarrow Q$ if function P calls function Q. For instance, Fig. 9.3 shows the calling graph associated with the merge sort algorithm of Section 2.9.

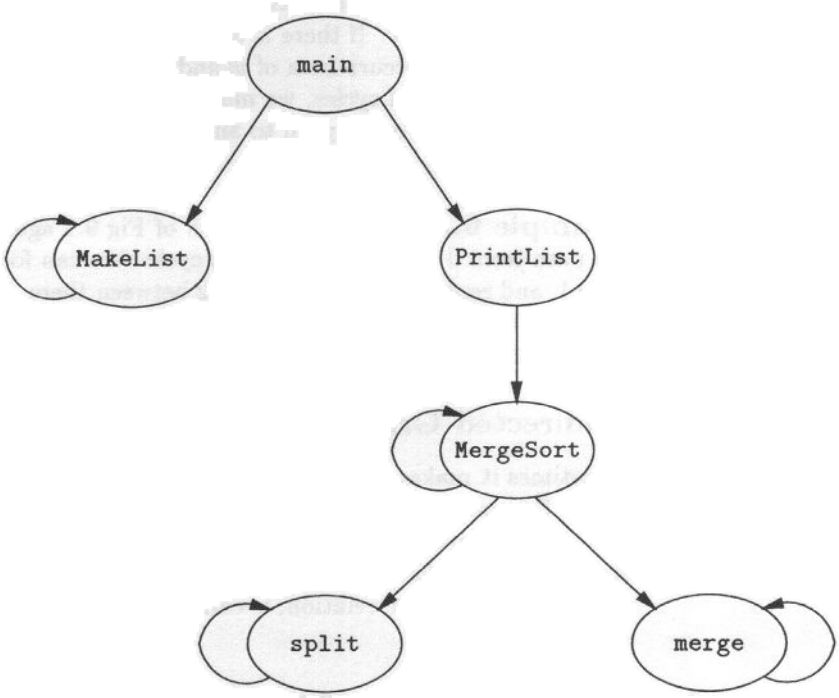
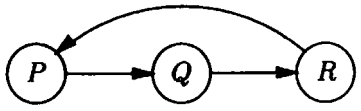


Fig. 9.3. Calling graph for the mergesort algorithm.

Direct and indirect recursion

The existence of a cycle in the calling graph implies a recursion in the algorithm. In Fig. 9.3 there are four simple cycles, one around each of the nodes **MakeList**, **MergeSort**, **split**, and **merge**. Each cycle is a trivial loop. Recall that all these functions call themselves, and thus are recursive. Recursions in which a function calls itself are by far the most common kind, and each of these appears as a loop in the calling graph. We call these recursions *direct*. However, one occasionally sees an *indirect* recursion, in which there is a cycle of length greater than 1. For instance, the graph



represents a function P that calls function Q, which calls function R, which calls function P. ◆

Acyclic Paths

A path is said to be *acyclic* if no node appears more than once on the path. Clearly, no cycle is acyclic. The argument that we just gave to show that for every cycle there is a simple cycle also demonstrates the following principle. If there is any path at all from u to v, then there is an acyclic path from u to v. To see why, start with any path from u to v. If there is a repetition of some node w, which could be u or v, replace the two occurrences of w and everything in between by one occurrence of w. As for the case of cycles, we may have to repeat this process several times, but eventually we reduce the path to an acyclic path.

✦ **Example 9.5.** Consider the graph of Fig 9.1 again. The path $(0, 1, 3, 2, 1, 3, 4)$ is a path from 0 to 4 that contains a cycle. We can focus on the two occurrences of node 1, and replace them, and the $3, 2$ between them, by 1, leaving $(0, 1, 3, 4)$, which is an acyclic path because no node appears twice. We could also have obtained the same result by focusing on the two occurrences of node 3. ✦

Undirected Graphs

Edge

Neighbors

Sometimes it makes sense to connect nodes by lines that have no direction, called *edges*. Formally, an edge is a set of two nodes. The edge $\{u, v\}$ says that nodes u and v are connected in both directions.[1] If $\{u, v\}$ is an edge, then nodes u and v are said to be *adjacent* or to be *neighbors*. A graph with edges, that is, a graph with a symmetric arc relation, is called an *undirected graph*.

✦ **Example 9.6.** Figure 9.4 represents a partial road map of the Hawaiian Islands, indicating some of the principal cities. Cities with a road between them are indicated by an edge, and the edge is labeled by the driving distance. It is natural to represent roads by edges, rather than arcs, because roads are normally two-way. ✦

Paths and Cycles in Undirected Graphs

A *path* in an undirected graph is a list of nodes $(v_1, v_2, \ldots, v_k)$ such that each node and the next are connected by an edge. That is, $\{v_i, v_{i+1}\}$ is an edge for $i = 1, 2, \ldots, k-1$. Note that edges, being sets, do not have their elements in any particular order. Thus, the edge $\{v_i, v_{i+1}\}$ could just as well appear as $\{v_{i+1}, v_i\}$.

The *length* of the path $(v_1, v_2, \ldots, v_k)$ is $k-1$. As with directed graphs, a node by itself is a path of length 0.

Defining cycles in undirected graphs is a little tricky. The problem is that we do not want to consider a path such as (u, v, u), which exists whenever there is an edge $\{u, v\}$, to be a cycle. Similarly, if $(v_1, v_2, \ldots, v_k)$ is a path, we can traverse it forward and backward, but we do not want to call the path

$$(v_1, v_2, \ldots, v_{k-1}, v_k, v_{k-1}, \ldots, v_2, v_1)$$

---

[1] Note that the edge is required to have exactly two nodes. A singleton set consisting of one node is not an edge. Thus, although an arc from a node to itself is permitted, we do not permit a looping edge from a node to itself. Some definitions of "undirected graph" do permit such loops.

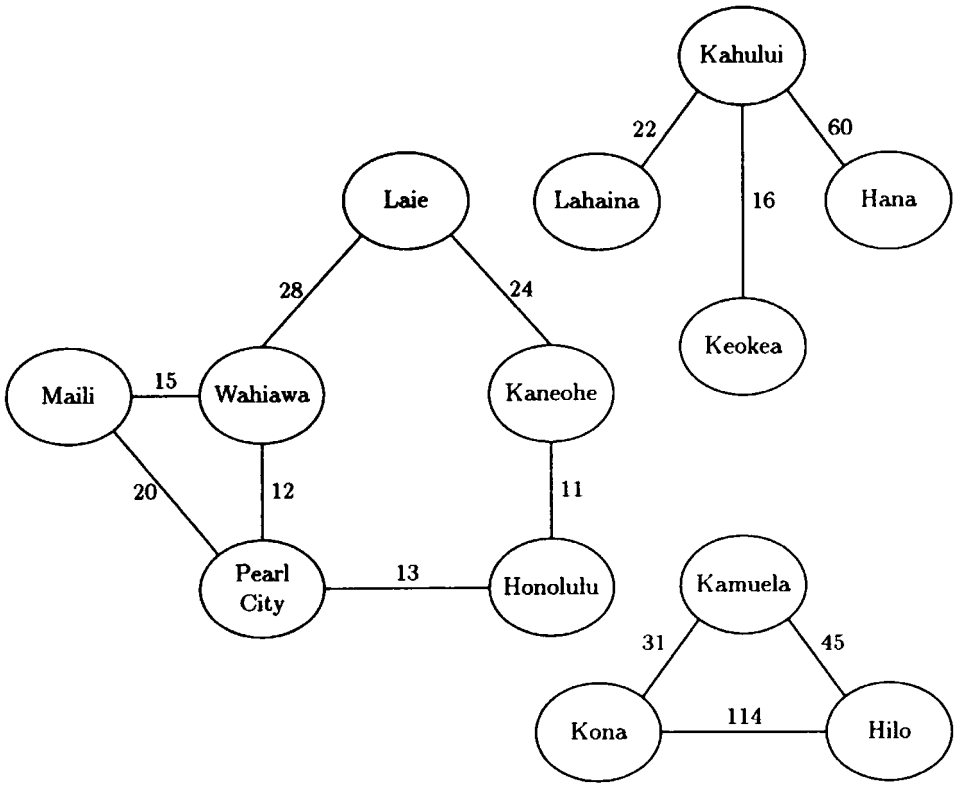


Fig. 9.4. An undirected graph representing roads in three Hawaiian Islands Oahu, Maui, and Hawaii (clockwise from the left).

a cycle.

Simple cycle

Perhaps the easiest approach is to define a *simple cycle* in an undirected graph to be a path of length three or more that begins and ends at the same node, and with the exception of the last node does not repeat any node. The notion of a nonsimple cycle in an undirected graph is not generally useful, and we shall not pursue this concept.

Equivalent cycles

As with directed cycles, we regard two undirected cycles as the same if they consist of the same nodes in the same order, with a different starting point. Undirected cycles are also the same if they consist of the same nodes in reverse order. Formally, the simple cycle $(v_1, v_2, \ldots, v_k)$ is equivalent, for each i between 1 and k, to the cycle $(v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_{i-1})$ and to the cycle

$$(v_i, v_{i-1}, \ldots, v_1, v_k, v_{k-1}, \ldots, v_{i+1})$$

✦ **Example 9.7.** In Fig. 9.4,

(Wahiawa, Pearl City, Maili, Wahiawa)

is a simple cycle of length three. It could have been written equivalently as

(Maili, Wahiawa, Pearl City, Maili)

by starting at Maili and proceeding in the same order around the circle. Likewise, it could have been written to start at Pearl City and proceed around the circle in reverse order:

(Pearl City, Maili, Wahiawa, Pearl City)

For another example,

(Laie, Wahiawa, Pearl City, Honolulu, Kaneohe, Laie)

is a simple cycle of length five. ✦

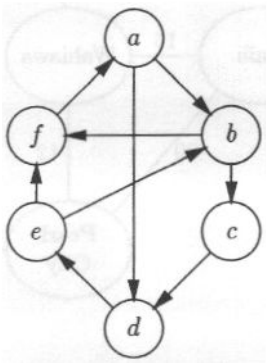


Fig. 9.5. Directed graph for Exercises 9.2.1 and 9.2.2.

EXERCISES

9.2.1: Consider the graph of Fig. 9.5.

a) How many arcs are there?
b) How many acyclic paths are there from node a to node d? What are they?
c) What are the predecessors of node b?
d) What are the successors of node b?
e) How many simple cycles are there? List them. Do not repeat paths that differ only in the starting point (see Exercise 9.2.8).
f) List all the nonsimple cycles of length up to 7.

9.2.2: Consider the graph of Fig. 9.5 to be an undirected graph, by replacing each arc $u \rightarrow v$ by an edge $\{u, v\}$.

a) Find all the paths from a to d that do not repeat any node.
b) How many simple cycles are there that include all six nodes? List these cycles.
c) What are the neighbors of node a?

9.2.3*: If a graph has 10 nodes, what is the largest number of arcs it can have? What is the smallest possible number of arcs? In general, if a graph has n nodes, what are the minimum and maximum number of arcs?

9.2.4*: Repeat Exercise 9.2.3 for the edges of an undirected graph.

9.2.5**: If a directed graph is acyclic and has n nodes, what is the largest possible number of arcs?

9.2.6: Find an example of indirect recursion among the functions so far in this book.

9.2.7: Write the cycle $(0, 1, 2, 0)$ in all possible ways.

9.2.8*: Let G be a directed graph and let R be the relation on the cycles of G defined by $(u_1, \ldots, u_k, u_1)R(v_1, \ldots, v_k, v_1)$ if and only if $(u_1, \ldots, u_k, u_1)$ and $(v_1, \ldots, v_k, v_1)$ represent the same cycle. Show that R is an equivalence relation on the cycles of G.

9.2.9*: Show that the relation S defined on the nodes of a graph by uSv if and only if $u = v$ or there is some cycle that includes both nodes u and v, is an equivalence relation.

9.2.10*: When we discussed simple cycles in undirected graphs, we mentioned that two cycles were really the same if they were the same nodes, either in order, or in reverse order, but with a different starting point. Show that the relation R consisting of pairs of representations for the same simple cycle is an equivalence relation.

9.3 Implementation of Graphs

There are two standard ways to represent a graph. One, called *adjacency lists*, is familiar from the implementation of binary relations in general. The second, called *adjacency matrices*, is a new way to represent binary relations, and is more suitable for relations where the number of pairs is a sizable fraction of the total number of pairs that could possibly exist over a given domain. We shall consider these representations, first for directed graphs, then for undirected graphs.

Adjacency Lists

Let nodes be named either by the integers $0, 1, \ldots, MAX - 1$ or by an equivalent enumerated type. In general, we shall use `NODE` as the type of nodes, but we may suppose that `NODE` is a synonym for `int`. Then we can use the generalized characteristic-vector approach, introduced in Section 7.9, to represent the set of arcs. This representation is called *adjacency lists*. We define linked lists of nodes by

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    LIST next;
};
```

and then create an array

```
LIST successors[MAX];
```

That is, the entry `successors[u]` contains a pointer to a linked list of all the successors of node u.

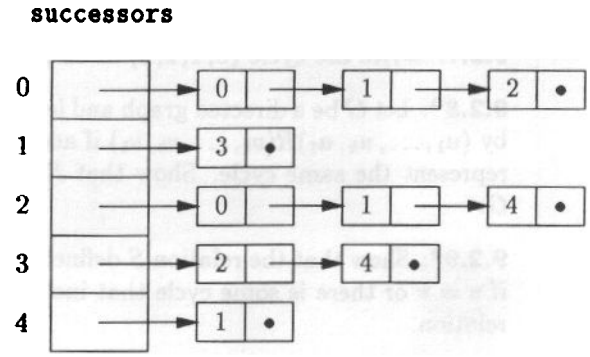


Fig. 9.6. Adjacency-list representation of the graph shown in Fig. 9.1.

✦ **Example 9.8.** The graph of Fig. 9.1 can be represented by the adjacency lists shown in Fig. 9.6. We have sorted the adjacency lists by node number, but the successors of a node can appear in any order on its adjacency list. ✦

Adjacency Matrices

Another common way to represent directed graphs is as *adjacency matrices*. We can create a two-dimensional array

```
BOOLEAN arcs[MAX][MAX];
```

in which the value of `arcs[u][v]` is `TRUE` if there is an arc $u \rightarrow v$, and `FALSE` if not.

✦ **Example 9.9.** The adjacency matrix for the graph of Fig. 9.1 is shown in Fig. 9.7. We use 1 for `TRUE` and 0 for `FALSE`. ✦

	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

Fig. 9.7. Adjacency matrix representing the graph of Fig. 9.1.

Operations on Graphs

We can see some of the distinctions between the two graph representations if we consider some simple operations on graphs. Perhaps the most basic operation is to determine whether there is an arc $u \rightarrow v$ from a node u to a node v. In the adjacency matrix, it takes $O(1)$ time to look up `arcs[u][v]` to see whether the entry there is `TRUE` or not.

Comparison of Adjacency Matrices and Adjacency Lists

Dense and sparse graphs

We tend to prefer adjacency matrices when the graphs are *dense*, that is, when the number of arcs is near the maximum possible number, which is n^2 for a graph of n nodes. However, if the graph is *sparse*, that is, if most of the possible arcs are not present, then the adjacency-list representation may save space. To see why, note that an adjacency matrix for an n-node graph has n^2 bits (provided we represent `TRUE` and `FALSE` by single bits rather than integers as we have done in this section).

In a typical computer, a structure consisting of an integer and a pointer, like our adjacency list cells, will use 32 bits to represent the integer and 32 bits to represent the pointer, or 64 bits total. Thus, if the number of arcs is a, we need about $64a$ bits for the lists, and $32n$ bits for the array of n headers. The adjacency list will use less space than the adjacency matrix if $32n + 64a < n^2$, that is, if $a < n^2/64 - n/2$. If n is large, we can neglect the $n/2$ term and approximate the previous inequality by $a < n^2/64$, that is, if fewer than 1/64th of the possible arcs are actually present. More detailed arguments favoring one or the other representation are presented when we discuss operations on graphs. The following table summarizes the preferred representations for various operations.

OPERATION	DENSE GRAPH	SPARSE GRAPH
Look up an arc	Adjacency matrix	Either
Find successors	Either	Adjacency lists
Find predecessors	Adjacency matrix	Either

With adjacency lists, it takes $O(1)$ time to find the header of the adjacency list for u. We must then traverse this list to the end if v is not there, or half the way down the list on the average if v is present. If there are a arcs and n nodes in the graph, then we take time $O(1 + a/n)$ on the average to do the lookup. If a is no more than a constant factor times n, this quantity is $O(1)$. However, the larger a is when compared with n, the longer it takes to tell whether an arc is present using the adjacency list representation. In the extreme case where a is around n^2, its maximum possible value, there are around n nodes on each adjacency list. In this case, it takes $O(n)$ time on the average to find a given arc. Put another way, the denser a graph is, the more we prefer the adjacency matrix to adjacency lists, when we need to look up a given arc.

On the other hand, we often need to find all the successors of a given node u. Using adjacency lists, we go to `successors[u]` and traverse the list, in average time $O(a/n)$, to find all the successors. If a is comparable to n, then we find all the successors of u in $O(1)$ time. But with adjacency matrices, we must examine the entire row for node u, taking $O(n)$ time no matter what a is. Thus, for graphs with a small number of edges per node, adjacency lists are much faster than adjacency matrices when we need to examine all the successors of a given node.

However, suppose we want to find all the predecessors of a given node v. With an adjacency matrix, we can examine the column for v; a 1 in the row for u means that u is a predecessor of v. This examination takes $O(n)$ time. The adjacency-list representation gives us no help finding predecessors. We must examine the adjacency list for every node u, to see if that list includes v. Thus, we may examine

A Matter of Degree

In- and Out-degree

The number of arcs out of a node v is called the *out-degree* of v. Thus, the out-degree of a node equals the length of its adjacency list; it also equals the number of 1's in the row for v in the adjacency matrix. The number of arcs into node v is the *in-degree* of v. The in-degree measures the number of times v appears on the adjacency list of some node, and it is the number of 1's found in the column for v in the adjacency matrix.

Degree of a graph

In an undirected graph, we do not distinguish between edges coming in or going out of a node. For an undirected graph, the *degree* of node v is the number of neighbors of v, that is, the number of edges $\{u, v\}$ containing v for some node u. Remember that in a set, order of members is unimportant, so $\{u, v\}$ and $\{v, u\}$ are the same edge, and are counted only once. The *degree of an undirected graph* is the maximum degree of any node in the graph. For example, if we regard a binary tree as an undirected graph, its degree is 3, since a node can only have edges to its parent, its left child, and its right child. For a directed graph, we can say that the *in-degree of a graph* is the maximum of the in-degrees of its nodes, and likewise, the *out-degree of a graph* is the maximum of the out-degrees of its nodes.

all the cells of all the adjacency lists, and we shall probably examine most of them. Since the number of cells in the entire adjacency list structure is equal to a, the number of arcs of the graph, the time to find predecessors using adjacency lists is thus $O(a)$ on a graph of a arcs. Here, the advantage goes to the adjacency matrix; and the denser the graph, the greater the advantage.

Implementing Undirected Graphs

Symmetric adjacency matrix

If a graph is undirected, we can pretend that each edge is replaced by arcs in both directions, and represent the resulting directed graph by either adjacency lists or an adjacency matrix. If we use an adjacency matrix, the matrix is *symmetric*. That is, if we call the matrix **edges**, then $edges[u][v] = edges[v][u]$. If we use an adjacency-list representation, then the edge $\{u, v\}$ is represented twice. We find v on the adjacency list for u and we find u on the list for v. That arrangement is often useful, since we cannot tell in advance whether we are more likely to follow the edge $\{u, v\}$ from u to v or from v to u.

	Laie	Kaneohe	Honolulu	PearlCity	Maili	Wahiawa
Laie	0	1	0	0	0	1
Kaneohe	1	0	1	0	0	0
Honolulu	0	1	0	1	0	0
PearlCity	0	0	1	0	1	1
Maili	0	0	0	1	0	1
Wahiawa	1	0	0	1	1	0

Fig. 9.8. Adjacency-matrix representation of an undirected graph from Fig. 9.4.

✦ **Example 9.10.** Consider how to represent the largest component of the undirected graph of Fig. 9.4 (which represents six cities on the island of Oahu). For the moment, we shall ignore the labels on the edges. The adjacency matrix representation is shown in Fig. 9.8. Notice that the matrix is symmetric.

Figure 9.9 shows the representation by adjacency lists. In both cases, we are using an enumeration type

```
enum CITYTYPE {Laie, Kaneohe, Honolulu,
    PearlCity, Maili, Wahiawa};
```

to index arrays. That arrangement is somewhat rigid, since it does not allow any changes in the set of nodes of the graph. We shall give a similar example shortly where we name nodes explicitly by integers, and use city names as node labels, for more flexibility in changing the set of nodes. ✦

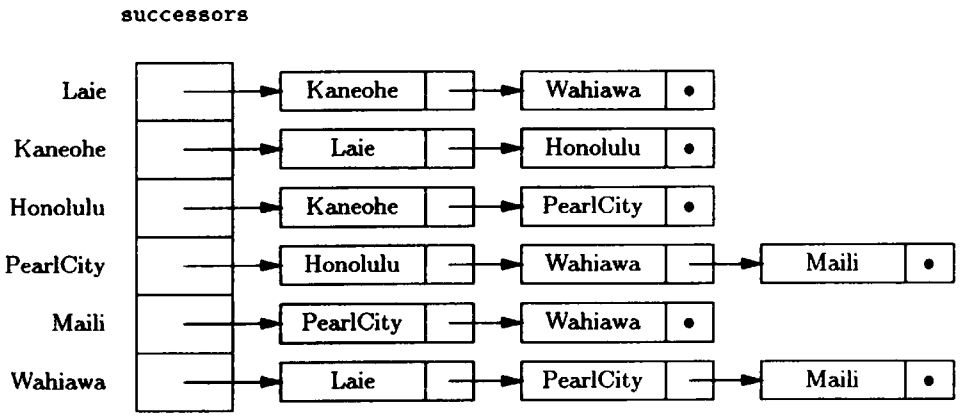


Fig. 9.9. Adjacency-list representation of an undirected graph from Fig. 9.4.

Representing Labeled Graphs

Suppose a graph has labels on its arcs (or edges if it is undirected). Using an adjacency matrix, we can replace the 1 that represents the presence of arc $u \rightarrow v$ in the graph by the label of this arc. It is necessary that we have some value that is permissible as a matrix entry but cannot be mistaken for a label; we use this value to represent the absence of an arc.

If we represent the graph by adjacency lists, we add to the cells forming the lists an additional field **nodeLabel**. If there is an arc $u \rightarrow v$ with label L, then on the adjacency list for node u we shall find a cell with v in its **nodeName** field and L in its **nodeLabel** field. That value represents the label of the arc.

We represent labels on nodes in a different way. For an adjacency matrix, we simply create another array, say **NodeLabels**, and let **NodeLabels[u]** be the label of node u. When we use adjacency lists, we already have an array of headers indexed by nodes. We change this array so that it has elements that are structures, one field for the node label and one field pointing to the beginning of the adjacency list.

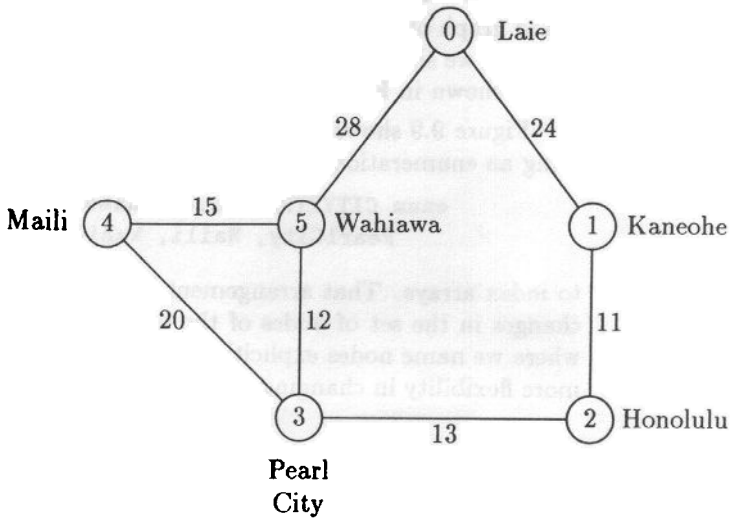


Fig. 9.10. Map of Oahu with nodes named by integers and labeled by cities.

cities

0	Laie
1	Kaneohe
2	Honolulu
3	PearlCity
4	Maili
5	Wahiawa

distances

	0	1	2	3	4	5
0	−1	24	−1	−1	−1	28
1	24	−1	11	−1	−1	−1
2	−1	11	−1		−1	−1
3	−1	−1		−1	20	12
4	−1	−1	−1		−1	15
5	28	−1	−1		15	−1

Fig. 9.11. Adjacency-matrix representation of a directed graph.

✦ **Example 9.11.** Let us again represent the large component of the graph of Fig. 9.4, but this time, we shall incorporate the edge labels, which are distances. Furthermore, we shall give the nodes integer names, starting with 0 for Laie, and proceeding clockwise. The city names themselves are indicated by node labels. We shall take the type of node labels to be character arrays of length 32. This representation is more flexible than that of Example 9.10, since if we allocate extra places in the array, we can add cities should we wish. The resulting graph is redrawn

in Fig. 9.10, and the adjacency matrix representation is in Fig. 9.11.

Notice that there are really two parts to this representation: the array `cities`, indicating the city that each of the integers 0 through 5 stands for, and the matrix `distances`, indicating the presence or absence of edges and the labels of present edges. We use -1 as a value that cannot be mistaken for a label, since in this example, labels, representing distances, must be positive.

We could declare this structure as follows:

```
typedef char CITYTYPE[32];
typedef CITYTYPE cities[MAX];
int distances[MAX][MAX];
```

Here, MAX is some number at least 6; it limits the number of nodes that can ever appear in our graph. `CITYTYPE` is defined to be 32-character arrays, and the array `cities` gives the labels of the various nodes. For example, we expect *cities*[0] to be `"Laie"`.

An alternative representation of the graph of Fig. 9.10 is by adjacency lists. Suppose the constant MAX and the type `CITYTYPE` are as above. We define the types `CELL` and `LIST` by

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    int distance;
    LIST next;
};
```

Next, we declare the array `cities` by

```
struct {
    CITYTYPE city;
    LIST adjacent;
} cities[MAX];
```

Figure 9.12 shows the graph of Fig. 9.10 represented in this manner. ✦

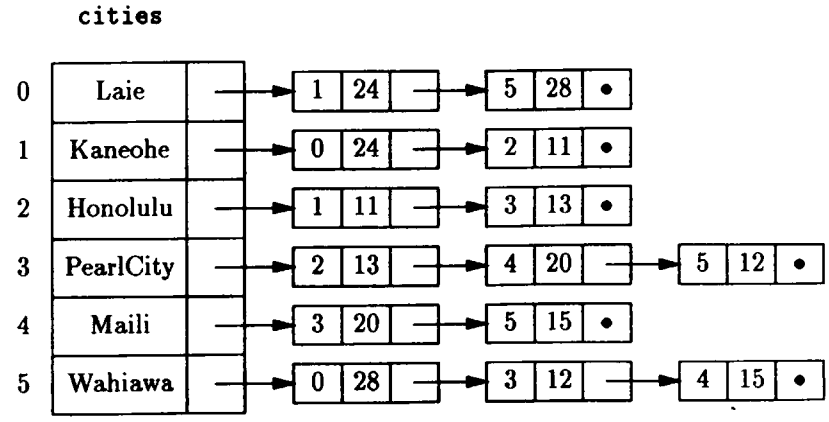


Fig. 9.12. Adjacency-list representation of graph with node and edge labels.

EXERCISES

9.3.1: Represent the graph of Fig. 9.5 (see the exercises of Section 9.2) by

a) Adjacency lists

b) An adjacency matrix

Give the appropriate type definitions in each case.

9.3.2: Suppose the arcs of Fig. 9.5 were instead edges (i.e., the graph were undirected). Repeat Exercise 9.3.1 for the undirected graph.

9.3.3: Let us label each of the arcs of the directed graph of Fig. 9.5 by the character string of length 2 consisting of the tail followed by the head. For example, the arc $a \rightarrow b$ is labeled by the character string `ab`. Also, suppose each node is labeled by the capital letter corresponding to its name. For instance, the node named a is labeled `A`. Repeat Exercise 9.3.1 for this labeled, directed graph.

9.3.4*: What is the relationship between the adjacency-matrix representation of an unlabeled graph and the characteristic-vector representation of a set of arcs?

9.3.5*: Prove by induction on n that in an undirected graph of n nodes, the sum of the degrees of the nodes is twice the number of edges. *Note*. A proof without using induction is also possible, but here an inductive proof is required.

9.3.6: Design algorithms to insert and delete arcs from an (a) adjacency-matrix (b) adjacency-list representation of a directed graph.

9.3.7: Repeat Exercise 9.3.6 for an undirected graph.

9.3.8: We can add a "predecessor list" to the adjacency-list representation of a directed or undirected graph. When is this representation preferred for the operations of

a) Looking up an arc?

b) Finding all successors?

c) Finding all predecessors?

Consider both dense and sparse graphs in your analysis.

9.4 Connected Components of an Undirected Graph

Connected graph

We can divide any undirected graph into one or more *connected components*. Each connected component is a set of nodes with paths from any member of the component to any other. Moreover, the connected components are maximal, that is, for no node in the component is there a path to any node outside the component. If a graph consists of a single connected component, then we say the graph is *connected*.

Physical Interpretation of Connected Components

If we are given a drawing of an undirected graph, it is easy to see the connected components. Imagine that the edges are strings. If we pick up any node, the connected component of which it is a member will come up with it, and members of all other connected components will stay where they are. Of course, what is easy to do by "eyeball" is not necessarily easy to do by computer. An algorithm to find the connected components of a graph is the principal subject of this section.

✦ **Example 9.12.** Consider again the graph of the Hawaiian Islands in Fig. 9.4. There are three connected components, corresponding to three islands. The largest component consists of Laie, Kaneohe, Honolulu, Pearl City, Maili, and Wahiawa. These are cities on the island of Oahu, and they are clearly mutually connected by roads, that is, by paths of edges. Also, clearly, there are no roads leading from Oahu to any other island. In graph-theoretic terms, there are no paths from any of the six cities mentioned above to any of the other cities in Fig. 9.4.

A second component consists of the cities of Lahaina, Kahului, Hana, and Keokea; these are cities on the island of Maui. The third component is the cities of Hilo, Kona, and Kamuela, on the "big island" of Hawaii. ✦

Connected Components as Equivalence Classes

Another useful way to look at connected components is that they are the equivalence classes of the equivalence relation P defined on the nodes of the undirected graph by: uPv if and only if there is a path from u to v. It is easy to check that P is an equivalence relation.

1. P is reflexive, that is, uPu for any node u, since there is a path of length 0 from any node to itself.
2. P is symmetric. If uPv, then there is a path from u to v. Since the graph is undirected, the reverse sequence of nodes is also a path. Thus vPu.
3. P is transitive. Suppose uPw and wPv are true. Then there is a path, say

   $$(x_1, x_2, \ldots, x_j)$$

   from u to w. Here, $u = x_1$ and $w = x_j$. Also, there is a path $(y_1, y_2, \ldots, y_k)$ from w to v where $w = y_1$ and $v = y_k$. If we put these paths together, we get a path from u to v, namely

   $$(u = x_1, x_2, \cdots, x_j = w = y_1, y_2, \cdots, y_k = v)$$

✦ **Example 9.13.** Consider the path

(Honolulu, PearlCity, Wahiawa, Maili)

from Honolulu to Maili in Fig. 9.10. Also consider the path

(Maili, PearlCity, Wahiawa, Laie)

from Maili to Laie in the same graph. If we put these paths together, we get a path from Honolulu to Laie:

(Honolulu, PearlCity, Wahiawa, Maili, PearlCity, Wahiawa, Laie)

It happens that this path is cyclic. As mentioned in Section 9.2, we can always remove cycles to get an acyclic path. In this case, one way to do so is to replace the two occurrences of Wahiawa and the nodes in between by one occurrence of Wahiawa to get

(Honolulu, PearlCity, Wahiawa, Laie)

which is an acyclic path from Honolulu to Laie. ✦

Since P is an equivalence relation, it partitions the set of nodes of the undirected graph in question into equivalence classes. The class containing node v is the set of nodes u such that vPu, that is, the set of nodes connected to v by a path. Moreover, another property of equivalence classes is that if nodes u and v are in different classes, then it is not possible that uPv; that is, there is never a path from a node in one equivalence class to a node in another. Thus, the equivalence classes defined by the "path" relation P are exactly the connected components of the graph.

An Algorithm for Computing the Connected Components

Suppose we want to construct the connected components of a graph G. One approach is to begin with a graph G_0 consisting of the nodes of G with none of the edges. We then consider the edges of G, one at a time, to construct a sequence of graphs $G_0, G_1, \ldots$, where G_i consists of the nodes of G and the first i edges of G.

BASIS. G_0 consists of only the nodes of G with none of the edges. Every node is in a component by itself.

INDUCTION. Suppose we have the connected components for the graph G_i after considering the first i edges, and we now consider the $(i+1)$st edge, $\{u, v\}$.

1. If u and v are in the same component of G_i, then G_{i+1} has the same set of connected components as G_i, because the new edge does not connect any nodes that were not already connected.

2. If u and v are in different components, we merge the components containing u and v to get the connected components for G_{i+1}. Figure 9.13 suggests why there is a path from any node x in the component of u, to any node y in the component of v. We follow the path in the first component from x to u, then the edge $\{u, v\}$, and finally the path from v to y that we know exists in the second component.

When we have considered all edges in this manner, we have the connected components of the full graph.

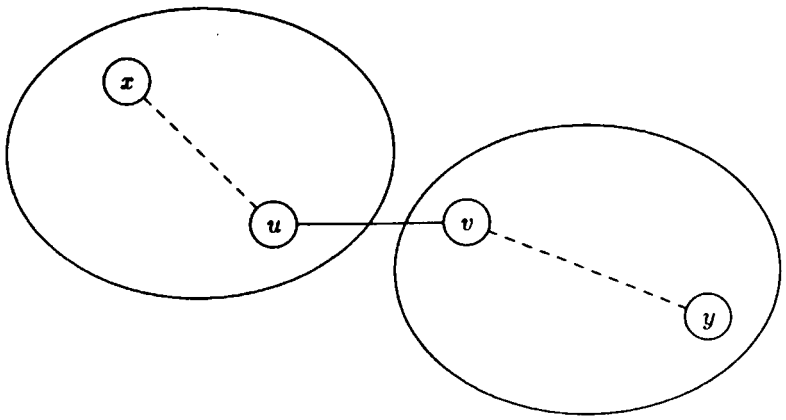


Fig. 9.13. Adding edge $\{u, v\}$ connects the components containing u and v.

✦ **Example 9.14.** Let us consider the graph of Fig. 9.4. We can consider edges in any order, but for reasons having to do with an algorithm in the next section, let us list the edges in order of the edge labels, smallest first. This list of edges is shown in Fig. 9.14.

Initially, all thirteen nodes are in components of their own. When we consider edge 1, {Kaneohe, Honolulu}, we merge these two nodes into a single component. The second edge, {Wahiawa, PearlCity}, merges those two cities. The third edge is {PearlCity, Honolulu}. That edge merges the components containing these two cities. Presently, each of these components contains two cities, so we now have one component with four cities, namely

{Wahiawa, PearlCity, Honolulu, Kaneohe}

All other cities are still in components by themselves.

EDGE	CITY 1	CITY 2	DISTANCE
1	Kaneohe	Honolulu	11
2	Wahiawa	PearlCity	12
3	PearlCity	Honolulu	13
4	Wahiawa	Maili	15
5	Kahului	Keokea	16
6	Maili	PearlCity	20
7	Lahaina	Kahului	22
8	Laie	Kaneohe	24
9	Laie	Wahiawa	28
10	Kona	Kamuela	31
11	Kamuela	Hilo	45
12	Kahului	Hana	60
13	Kona	Hilo	114

Fig. 9.14. Edges of Fig. 9.4 in order of labels.

Edge 4 is {Maili, Wahiawa} and adds Maili to the large component. The fifth edge is {Kahului, Keokea}, which merges these two cities into a component. When we consider edge 6, {Maili, PearlCity}, we see a new phenomenon: both ends of the edge are already in the same component. We therefore do no merging with edge 6.

Edge 7 is {Lahaina, Kahului}, and it adds the node Lahaina to the component {Kahului, Keokea}, forming the component {Lahaina, Kahului, Keokea}. Edge 8 adds Laie to the largest component, which is now

{Laie, Kaneohe, Honolulu, PearlCity, Wahiawa, Maili}

The ninth edge, {Laie, Wahiawa}, connects two cities in this component and is thus ignored.

Edge 10 groups Kamuela and Kona into a component, and edge 11 adds Hilo to this component. Edge 12 adds Hana to the component of

{Lahaina, Kahului, Keokea}

Finally, edge 13, {Hilo, Kona}, connects two cities already in the same component. Thus,

{Laie, Kaneohe, Honolulu, PearlCity, Wahiawa, Maili}
{Lahaina, Kahului, Keokea, Hana}
{Kamuela, Hilo, Kona}

is the final set of connected components. ✦

A Data Structure for Forming Components

If we consider the algorithm described informally above, we need to be able to do two things quickly:

1. Given a node, find its current component.

2. Merge two components into one.

There are a number of data structures that can support these operations. We shall study one simple idea that gives surprisingly good performance. The key is to put the nodes of each component into a tree.[2] The component is represented by the root of the tree. The two operations above can now be implemented as follows:

1. To find the component of a node in the graph, we go to the representative of that node in the tree and follow the path in that tree to the root, which represents the component.

2. To merge two different components, we make the root of one component a child of the root of the other.

---

[2] It is important to understand that, in what follows, the "tree" and the "graph" are distinct structures. There is a one-to-one correspondence between the nodes of the graph and the nodes of the tree; that is, each tree node represents a graph node. However, the parent-child edges of the tree are not necessarily edges in the graph.

✦ **Example 9.15.** Let us follow the steps of Example 9.14, showing the trees created at certain steps. Initially, every node is in a one-node tree by itself. The first edge, {Kaneohe, Honolulu}, causes us to merge two one-node trees, {Kaneohe} and {Honolulu}, into one two-node tree, {Kaneohe, Honolulu}. Either node could be made a child of the other. Let us suppose that Honolulu is made the child of the root Kaneohe.

Similarly, the second edge, {Wahiawa, PearlCity}, merges two trees, and we may suppose that PearlCity is made the child of the root Wahiawa. At this point, the current collection of components is represented by the two trees in Fig. 9.15 and nine one-node trees.

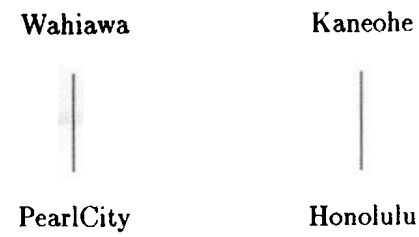


Fig. 9.15. The first two nontrivial trees as we merge components.

The third edge, {PearlCity, Honolulu}, merges these two components. Let us suppose that Wahiawa is made a child of the other root, Kaneohe. Then the resulting component is represented by the tree of Fig. 9.16.

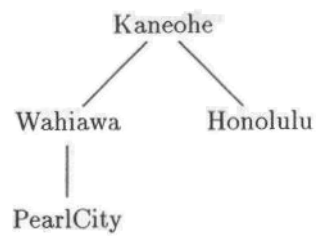


Fig. 9.16. Tree representing component of four nodes.

When we consider the fourth edge, {Wahiawa, Maili}, we merge Maili into the component represented by the tree of Fig. 9.16. We could either make Maili a child of Kaneohe, or make Kaneohe a child of Maili. We prefer the former, since that keeps the height of the tree small, while making the root of the large component a child of the root of the small component tends to make paths in the tree larger. Large paths, in turn, cause us to take more time following a path to the root, which we need to do to determine the component of a node. By following that policy and making arbitrary decisions when components have the same height, we might wind up with the three trees in Fig. 9.17 that represent the three final connected components. ✦

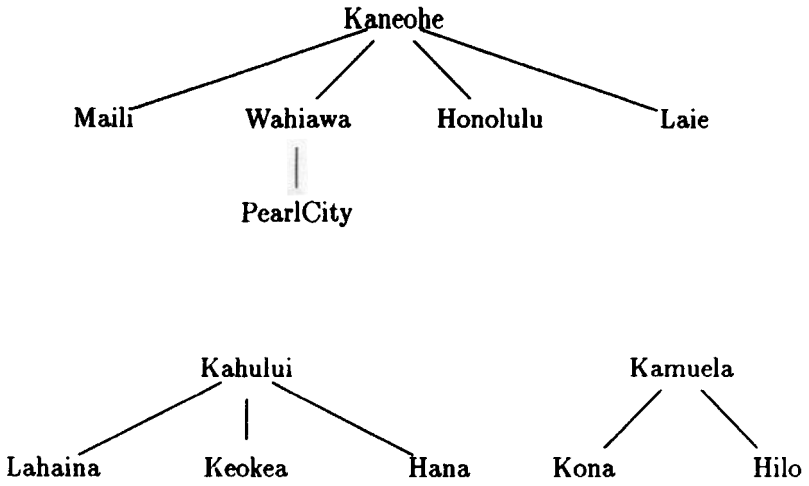


Fig. 9.17. Trees representing final connected components using tree-merging algorithm.

Following the lesson of Example 9.15, we formulate a policy that whenever we merge two trees, the root of lesser height becomes a child of the root with greater height. Ties can be broken arbitrarily. The important gain from this policy is that heights of trees can only grow logarithmically with the number of nodes in the trees, and in practice, the height is often smaller. Therefore, when we follow a path from a tree node to its root, we take at most time proportional to the logarithm of the number of nodes in the tree. We can derive the logarithmic bound by proving the following statement by induction on the height h.

STATEMENT $S(h)$: A tree of height h, formed by the policy of merging lower into higher, has at least 2^h nodes.

BASIS. The basis is $h = 0$. Such a tree must be a single node, and since $2^0 = 1$, the statement $S(0)$ is true.

INDUCTION. Suppose $S(h)$ is true for some $h \geq 0$, and consider a tree T of height $h+1$. At some time during the formation of T by mergers, the height first reached $h+1$. The only way to get a tree of height $h+1$ is to make the root of some tree T_1, of height h, a child of the root of some tree T_2. T is T_1 plus T_2, plus perhaps other nodes that were added later, as suggested by Fig. 9.18.

Now T_1, by the inductive hypothesis, has at least 2^h nodes. Since its root was made a child of the root of T_2, the height of T_2 is also at least h. Thus, T_2 also has at least 2^h nodes. T consists of T_1, T_2, and perhaps more, so T has at least $2^h + 2^h = 2^{h+1}$ nodes. That statement is $S(h+1)$, and we have proved the inductive step.

We now know that if a tree has n nodes and height h, it must be that $n \geq 2^h$. Taking logarithms of both sides, we have $\log_2 n \geq h$; that is, the height of the tree

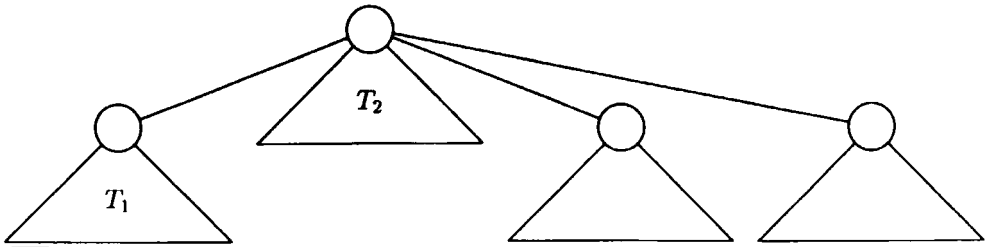


Fig. 9.18. Forming a tree of height $h + 1$.

cannot be greater than the logarithm of the number of nodes. Consequently, when we follow any path from a node to its root, we take $O(\log n)$ time.

We shall now describe in more detail the data structure that implements these ideas. First, suppose that there is a type **NODE** representing nodes. As before, we assume the type **NODE** is **int** and **MAX** is at least the number of nodes in the graph. For our example of Fig. 9.4, we shall let **MAX** be 13.

We shall also assume that there is a list **edges** consisting of cells of type **EDGE**. These cells are defined by

```
typedef struct EDGE *EDGELIST;
struct EDGE {
    NODE node1, node2;
    EDGELIST next;
};
```

Finally, for each node of the graph, we need a corresponding tree node. Tree nodes will be structures of type **TREENODE**, consisting of

1. A parent pointer, so that we can build a tree on the graph's nodes, and follow the tree to its root. A root node will be identified by having **NULL** as its parent.
2. The height of the tree of which a given node is the root. The height will only be used if the node is presently a root.

We may thus define type **TREENODE** by

```
typedef struct TREENODE *TREE;
struct TREENODE {
    int height;
    TREE parent;
};
```

We shall define an array

```
TREE nodes[MAX];
```

to associate with each graph node a node in some tree. It is important to realize that each entry in the array **nodes** is a pointer to a node in the tree, yet this entry is the sole representative of the node in the graph.

Two important auxiliary functions are shown in Fig. 9.19. The first, **find**, takes a node a, gets a pointer to the corresponding tree node, x, and follows the parent pointers in x and its ancestors, until it comes to the root. This search for the root is performed by lines (2) and (3). If the root is found, a pointer to the root is returned at line (4). Note that at line (1), the type **NODE** must be **int** so it may

```
     /* return the root of the tree containing the tree node x
        corresponding to graph node a */
     TREE find(NODE a, TREE nodes[]);
     {
         TREE x;

(1)      x = nodes[a];
(2)      while (x->parent != NULL)
(3)          x = x->parent;
(4)      return x;
     }


     /* merge the trees with roots x and y into one tree,
           by making the root of the lower a child of
           the root of the higher */
     void merge(TREE x, TREE y)
     {
         TREE higher, lower;

(5)      if (x->height > y->height) {
(6)          higher = x;
(7)          lower = y;
         }
         else {
(8)          higher = y;
(9)          lower = x;
         }
(10)     lower->parent = higher;
(11)     if (lower->height == higher->height)
(12)         ++(higher->height);
     }
```

Fig. 9.19. Auxiliary functions `find` and `merge`.

be used to index the array `nodes`.

The second function, `merge`,[3] takes pointers to two tree nodes, x and y, which must be the roots of distinct trees for the merger to work properly. The test of line (5) determines which of the roots has the greater height; ties are broken in favor of y. The higher is assigned to the local variable `higher` and the lower to the local variable `lower` at lines (6–7) or lines (8–9), whichever is appropriate. Then at line (10) the lower is made a child of the higher and at lines (11) and (12) the height of the higher, which is now the root of the combined tree, is incremented by one if the heights of T_1 and T_2 are equal. The height of the lower remains as it was, but it is now meaningless, because the lower is no longer a root.

The heart of the algorithm to find connected components is shown in Fig. 9.20.

[3] Do not confuse this function with a function of the same name used for merge sorting in Chapters 2 and 3.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 13
typedef int NODE;
typedef struct EDGE *EDGELIST;
struct EDGE {
    NODE node1, node2;
    EDGELIST next;
};

typedef struct TREENODE *TREE
struct TREENODE {
    int height;
    TREE parent;
};

TREE find(NODE a, TREE nodes[]);
void merge(TREE x, TREE y);
EDGELIST makeEdges();

main()
{
    NODE u;
    TREE a, b;
    EDGELIST e;
    TREE nodes[MAX];

    /* initialize nodes so each node is in a tree by itself */
(1) for (u = 0; u < MAX; u++) {
(2)     nodes[u] = (TREE) malloc(sizeof(struct TREENODE));
(3)     nodes[u]->parent = NULL;
(4)     nodes[u]->height = 0;
    }

    /* initialize e as the list of edges of the graph */
(5) e = makeEdges();

    /* examine each edge, and if its ends are in different
       components, then merge them */
(6) while (e != NULL) {
(7)     a = find(e->node1, nodes);
(8)     b = find(e->node2, nodes);
(9)     if (a != b)
(10)        merge(a, b);
(11)    e = e->next;
    }
}
```

Fig. 9.20. C program to find connected components.

Better Algorithms for Connected Components

We shall see, when we learn about depth-first search in Section 9.6, that there is actually a better way to compute connected components, one that takes only $O(m)$ time, instead of $O(m \log n)$ time. However, the data structure given in Section 9.4 is useful in its own right, and we shall see in Section 9.5 another program that uses this data structure.

We assume that the function **makeEdges()** turns the graph at hand into a list of edges. The code for this function is not shown.

Lines (1) through (4) of Fig. 9.20 go down the array **nodes**, and for each node, a tree node is created at line (2). Its **parent** field is set to **NULL** at line (3), making it the root of its own tree, and its **height** field is set to 0 at line (4), reflecting the fact that the node is alone in its tree.

Line (5) then initializes **e** to point to the first edge on the list of edges, and the loop of lines (6) through (11) examines each edge in turn. At lines (7) and (8) we find the roots of the two ends of the current edge. Then at line (9) we test to see if these roots are different tree nodes. If so, the ends of the current edge are in different components, and we merge these components at line (10). If the two ends of the edge are in the same component, we skip line (10), so no change to the collection of trees is made. Finally, line (11) advances us along the list of edges.

Running Time of the Connected Components Algorithm

Let us determine how long the algorithm of Fig. 9.20 takes to process a graph. Suppose the graph has n nodes, and let m be the larger of the number of nodes and the number of edges.[4] First, let us examine the auxiliary functions. We argued that the policy of merging lower trees into higher ones guarantees that the path from any tree node to its root cannot be longer than $\log n$. Thus, **find** takes $O(\log n)$ time.

Next, let us examine the function **merge** from Fig. 9.19. Each of its statements takes $O(1)$ time. Since there are no loops or function calls, the entire function takes $O(1)$ time.

Finally, let us examine the main program of Fig. 9.20. The body of the for-loop of lines (1) to (4) takes $O(1)$ time, and the loop is iterated n times. Thus, the time for lines (1) through (4) is $O(n)$. Let us assume line (5) takes $O(m)$ time. Finally, consider the while-loop of lines (6) to (11). In the body, lines (7) and (8) each take $O(\log n)$ time, since they are calls to a function, **find**, that we just determined takes $O(\log n)$ time. Lines (9) and (11) clearly take $O(1)$ time. Line (10) likewise takes $O(1)$ time, because we just determined that function **merge** takes $O(1)$ time. Thus, the entire body takes $O(\log n)$ time. The while-loop iterates m times, where m is the number of edges. Thus, the time for this loop is $O(m \log n)$, that is, the number of iterations times the bound on the time for the body.

In general, then, the running time of the entire program can be expressed as $O(n + m + m \log n)$. However, m is at least n, and so the $m \log n$ term dominates the other terms. Thus, the running time of the program in Fig. 9.20 is $O(m \log n)$.

[4] It is normal to think of m as the number of edges, but in some graphs, there are more nodes than edges.

CITY 1	CITY 2	DISTANCE
Marquette	Sault Ste. Marie	153
Saginaw	Flint	31
Grand Rapids	Lansing	60
Detroit	Lansing	78
Escanba	Sault Ste. Marie	175
Ann Arbor	Detroit	28
Ann Arbor	Battle Creek	89
Battle Creek	Kalamazoo	21
Menominee	Escanba	56
Kalamazoo	Grand Rapids	45
Escanba	Marquette	78
Battle Creek	Lansing	40
Flint	Detroit	58

Fig. 9.21. Some distances within the state of Michigan.

EXERCISES

9.4.1: Figure 9.21 lists some cities in the state of Michigan and the road mileage between them. For the purposes of this exercise, ignore the mileage. Construct the connected components of the graph by examining each edge in the manner described in this section.

9.4.2*: Prove, by induction on k, that a connected component of k nodes has at least $k - 1$ edges.

9.4.3*: There is a simpler way to implement "merge" and "find," in which we keep an array indexed by nodes, giving the component of each node. Initially, each node is in a component by itself, and we name the component by the node. To find the component of a node, we simply look up the corresponding array entry. To merge components, we run down the array, changing each occurrence of the first component to the second.

a) Write a C program to implement this algorithm.

b) As a function of n, the number of nodes, and m, the larger of the number of nodes and edges, what is the running time of this program?

c) For certain numbers of edges and nodes, this implementation is actually better than the one described in the section. When?

9.4.4*: Suppose that instead of merging lower trees into higher trees in the connected components algorithm of this section, we merge trees with fewer nodes into trees with a larger number of nodes. Is the running time of the connected-components algorithm still $O(m \log n)$?

9.5 Minimal Spanning Trees

There is an important generalization of the connected components problem, in which we are given an undirected graph with edges labeled by numbers (integers or reals). We must not only find the connected components, but for each component we must find a tree connecting the nodes of that component. Moreover, this tree must be *minimal*, meaning that the sum of the edge labels is as small as possible.

Unrooted, unordered trees

The trees talked about here are not quite the same as the trees of Chapter 5. Here, no node is designated the root, and there is no notion of children or of order among the children. Rather, when we speak of "trees" in this section, we mean unrooted, unordered trees, which are just undirected graphs that have no simple cycles.

Spanning tree

A *spanning tree* for an undirected graph G is the nodes of G together with a subset of the edges of G that

1. Connect the nodes; that is, there is a path between any two nodes using only edges in the spanning tree.
2. Form an unrooted, unordered tree; that is, there are no (simple) cycles.

If G is a single connected component, then there is always a spanning tree. A *minimal spanning tree* is a spanning tree the sum of whose edge labels is as small as that of any spanning tree for the given graph.

✦ **Example 9.16.** Let graph G be the connected component for the island of Oahu, as in Fig. 9.4 or Fig. 9.10. One possible spanning tree is shown in Fig. 9.22. It is formed by deleting the edges {Maili, Wahiawa} and {Kaneohe, Laie}, and retaining the other five edges. The *weight*, or sum of edge labels, for this tree is 84. As we shall see, that is not a minimum. ✦

Weight of a tree

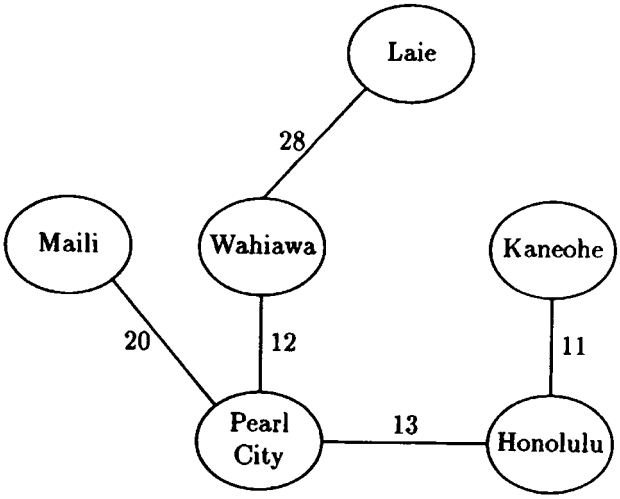
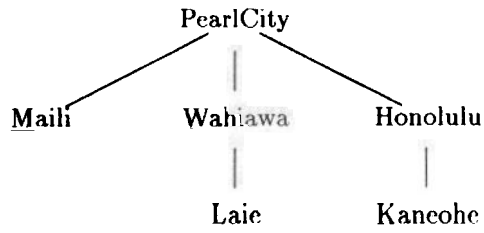


Fig. 9.22. A spanning tree for the island of Oahu.

Rooted and Unrooted Trees

The notion of an unrooted tree should not seem too strange. In fact, we can choose any node of an unrooted tree to be the root. That gives a direction to all edges, away from the root, or from parent to child. Physically, it is as if we picked up the unrooted tree by a node, letting the rest of the tree dangle from the selected node. For example, we could make Pearl City the root of the spanning tree in Fig. 9.22, and it would look like this:



We can order the children of each node if we wish, but the order will be arbitrary, bearing no relation to the original unrooted tree.

Finding a Minimal Spanning Tree

Kruskal's algorithm

There are a number of algorithms to find minimal spanning trees. We shall exhibit one, called *Kruskal's algorithm,* that is a simple extension to the algorithm discussed in the last section for finding connected components. The changes needed are

1. We are required to consider edges in increasing order of their labels. (We happened to choose that order in Example 9.14, but it was not required for connected components.)
2. As we consider edges, if an edge has its ends in different components, then we select that edge for the spanning tree and merge components, as in the algorithm of the previous section. Otherwise, we do not select the edge for the spanning tree, and, of course, we do not merge components.

✦ **Example 9.17.** The Acme Surfboard Wax Company has offices in the thirteen cities shown in Fig. 9.4. It wishes to rent dedicated data transmission lines from the phone company, and we shall suppose that the phone lines run along the roads that are indicated by edges in Fig. 9.4. Between islands, the company must use satellite transmission, and the cost will be proportional to the number of components. However, for the ground transmission lines, the phone company charges by the mile.[5] Thus, we wish to find a minimal spanning tree for each connected component of the graph of Fig. 9.4.

If we divide the edges by component, then we can run Kruskal's algorithm on

[5] This is one possible way to charge for leased telephone lines. One finds a minimal spanning tree connecting the desired sites, and the charge is based on the weight of that tree, regardless of how the phone connections are provided physically.

each component separately. However, if we do not already know the components, then we must consider all the edges together, smallest label first, in the order of Fig. 9.14. As in Section 9.4, we begin with each node in a component by itself.

We first consider the edge {Kaneohe, Honolulu}, the edge with the smallest label. This edge merges these two cities into one component, and because we perform a merge operation, we select that edge for the minimal spanning tree. Edge 2 is {Wahiawa, PearlCity}, and since that edge also merges two components, it is selected for the spanning tree. Likewise, edges 3 and 4, {PearlCity, Honolulu} and {Wahiawa, Maili}, merge components, and are therefore put in the spanning tree.

Edge 5, {Kahului, Keokea}, merges these two cities, and is also accepted for the spanning tree, although this edge will turn out to be part of the spanning tree for the Maui component, rather than the Oahu component as was the case for the four previous edges.

Edge 6, {Maili, PearlCity}, connects two cities that are already in the same component. Thus, this edge is rejected for the spanning tree. Even though we shall have to pick some edges with larger labels, we cannot pick {Maili, PearlCity}, because to do so would form a cycle of the cities Maili, Wahiawa, and Pearl City. We cannot have a cycle in the spanning tree, so one of the three edges must be excluded. As we consider edges in order of label, the last edge of the cycle considered must have the largest label, and is the best choice to exclude.

Edge 7, {Lahaina, Kahului}, and edge 8, {Laie, Kaneohe}, are both accepted for the spanning tree, because they merge components. Edge 9, {Laie, Wahiawa}, is rejected because its ends are in the same component. We accept edges 10 and 11; they form the spanning tree for the "big island" component, and we accept edge 12 to complete the Maui component. Edge 13 is rejected, because it connects Kona and Hilo, which were merged into the same component by edges 10 and 11. The resulting spanning trees of the components are shown in Fig. 9.23. ✦

Why Kruskal's Algorithm Works

We can prove that Kruskal's algorithm produces a spanning tree whose weight is as small as that of any spanning tree for the given graph. Let G be an undirected, connected graph. For convenience, let us add infinitesimal amounts to some labels, if necessary, so that all labels are distinct, and yet the sum of the added infinitesimals is not as great as the difference between two edges of G that have different labels. As a result, G with the new labels will have a unique minimal spanning tree, which will be one of the minimal spanning trees of G with the original weights.

Then, let $e_1, e_2, \ldots, e_m$ be all the edges of G, in order of their labels, smallest first. Note that this order is also the order in which Kruskal's algorithm considers the edges. Let K be the spanning tree for G with the adjusted labels produced by Kruskal's algorithm, and let T be the unique minimal spanning tree for G.

We shall prove that K and T are really the same. If they are different, then there must be at least one edge that is in one but not the other. Let e_i be the first such edge in the ordering of edges; that is, each of $e_1, \ldots, e_{i-1}$ is either in both K and T, or in neither of K and T. There are two cases, depending on whether e_i is in K or is in T. We shall show a contradiction in each case, and thus conclude that e_i does not exist; thus $K = T$, and K is the minimal spanning tree for G.

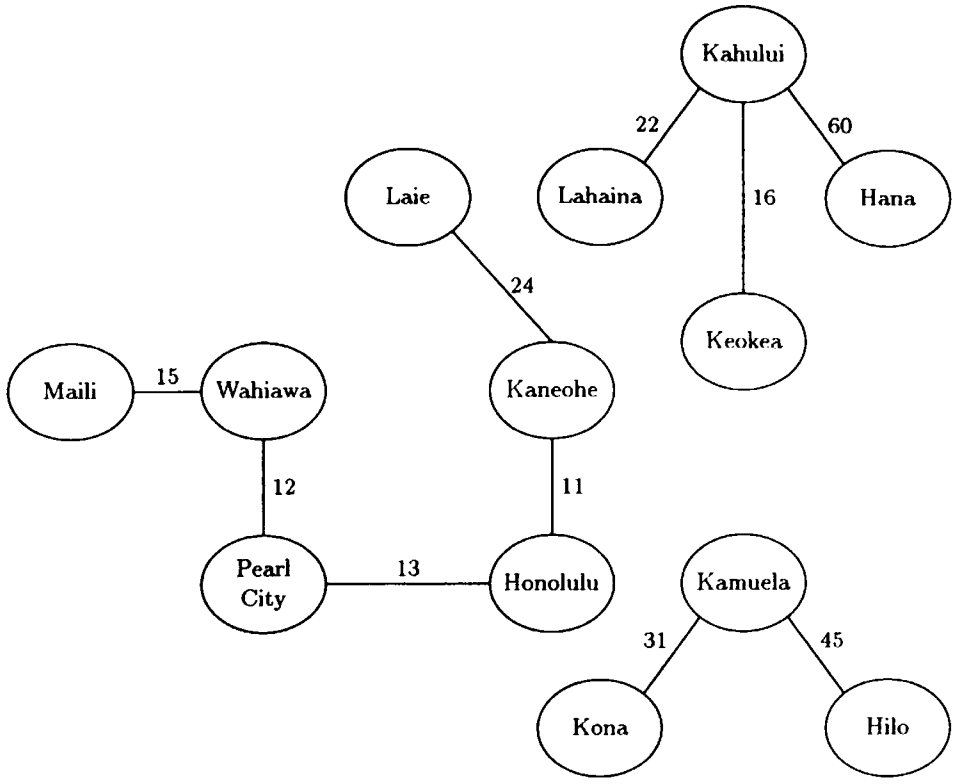


Fig. 9.23. Spanning trees for the graph of Fig. 9.4.

Greed Sometimes Pays

Greedy algorithm

Kruskal's algorithm is a good example of a *greedy algorithm*, in which we make a series of decisions, each doing what seems best at the time. The local decisions are which edge to add to the spanning tree being formed. In each case, we pick the edge with the least label that does not violate the definition of "spanning tree" by completing a cycle. Often, the overall effect of locally optimal decisions is not globally optimum. However, in the case of Kruskal's algorithm, it can be shown that the result is globally optimal; that is, a spanning tree of minimal weight results.

Case 1. Edge e_i is in T but not in K. If Kruskal's algorithm rejects e_i, then e_i must form a cycle with some path P of edges previously selected for K, as suggested in Fig. 9.24. Thus, the edges of P are all found among $e_1, \ldots, e_{i-1}$. However, T and K agree about these edges; that is, if the edges of P are in K, then they are also in T. But since T has e_i as well, P plus e_i form a cycle in T, contradicting our assumption that T was a spanning tree. Thus, it is not possible that e_i is in T but not in K.

Case 2. Edge e_i is in K but not in T. Let e_i connect the nodes u and v. Since T is connected, there must be some acyclic path in T between u and v; call it path Q. Since Q does not use edge e_i, Q plus e_i forms a simple cycle in the graph G.

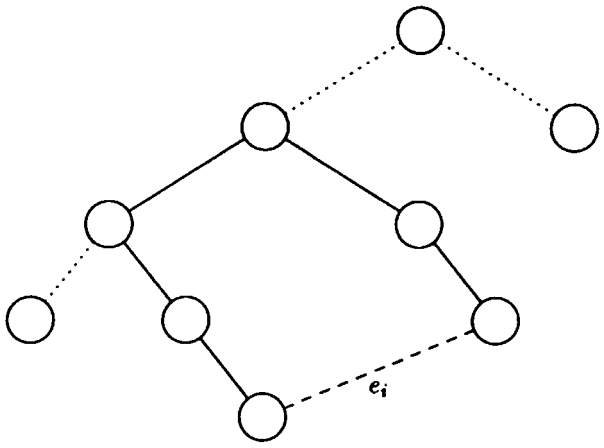


Fig. 9.24. Path P (solid lines) is in T and K; edge e_i is in T only.

There are two subcases, depending on whether or not e_i has a higher label than all the edges on path Q.

a) Edge e_i has the highest label. Then all the edges on Q are among $\{e_1, \ldots, e_{i-1}\}$. Remember that T and K agree on all edges before e_i, and so all the edges of Q are also edges of K. But e_i is also in K, which implies K has a cycle. We thus rule out the possibility that e_i has a higher label than any of the edges of path Q.

b) There is some edge f on path Q that has a higher label than e_i. Suppose f connects nodes w and x. Figure 9.25 shows the situation in tree T. If we remove edge f from T, and add edge e_i, we do not form a cycle, because path Q was broken by the removal of f. The resulting collection of edges has a lower weight than T, because f has a higher label than e_i. We claim the resulting edges still connect all the nodes. To see why, notice that w and x are still connected; there is a path that follows Q from w to u, then follows the edge e_i, then the path Q from v to x. Since $\{w, x\}$ was the only edge removed, if its endpoints are still connected, surely all nodes are connected. Thus, the new set of edges is a spanning tree, and its existence contradicts the assumption that T was minimal.

We have now shown that it is impossible for e_i to be in K but not in T. That rules out the second case. Since it is impossible that e_i is in one of T and K, but not the other, we conclude that K really is the minimal spanning tree T. That is, Kruskal's algorithm always finds a minimal spanning tree.

Running Time of Kruskal's Algorithm

Suppose we run Kruskal's algorithm on a graph of n nodes. As in the previous section, let m be the larger of the number of nodes and the number of edges, but remember that typically the number of edges is the larger. Let us suppose that the graph is represented by adjacency lists, so we can find all the edges in $O(m)$ time.

To begin, we must sort the edges by label, which takes $O(m \log m)$ time, if we use an efficient sorting algorithm such as merge sort. Next, we consider the edges, taking $O(m \log n)$ time to do all the merges and finds, as discussed in the

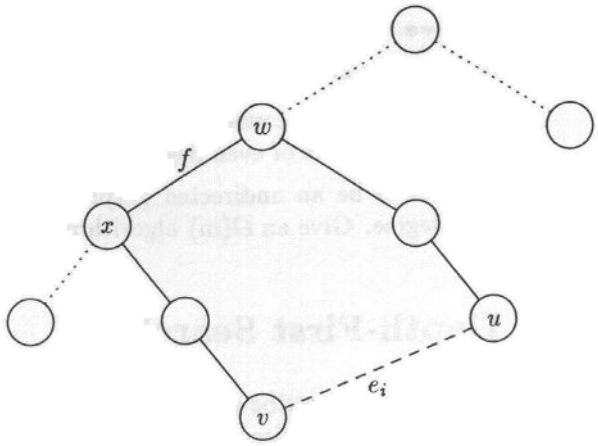


Fig. 9.25. Path Q (solid) is in T.
We can add edge e_i to T and remove the edge f.

previous section. It appears that the total time for Kruskal's algorithm is thus $O(m(\log n + \log m))$.

However, notice that $m \leq n^2$, because there are only $n(n-1)/2$ pairs of nodes. Thus, $\log m \leq 2 \log n$, and $m(\log n + \log m) \leq 3m \log n$. Since constant factors can be neglected within a big-oh expression, we conclude that Kruskal's algorithm takes $O(m \log n)$ time.

EXERCISES

9.5.1: Draw the tree of Fig. 9.22 if Wahiawa is selected as the root.

9.5.2: Use Kruskal's algorithm to find minimal spanning trees for each of the components of the graph whose edges and labels are listed in Fig. 9.21 (see the exercises for Section 9.4).

9.5.3**: Prove that if G is a connected, undirected graph of n nodes, and T is a spanning tree for G, then T has $n-1$ edges. *Hint*: We need to do an induction on n. The hard part is to show that T must have some node v with degree 1; that is, T has exactly one edge containing v. Consider what would happen if for every node u, there were at least two edges of T containing u. By following edges into and out of a sequence of nodes, we would eventually find a cycle. Since T is supposedly a spanning tree, it could not have a cycle, which gives us a contradiction.

9.5.4*: Once we have selected $n-1$ edges, it is not necessary to consider any more edges for possible inclusion in the spanning tree. Describe a variation of Kruskal's algorithm that does not sort all the edges, but puts them in a priority queue, with the negative of the edge's label as its priority (i.e., shortest edge is selected first by *deleteMax*). Show that if a spanning tree can be found among the first $m/\log m$ edges, then this version of Kruskal's algorithm takes only $O(m)$ time.

9.5.5*: Suppose we find a minimal spanning tree T for a graph G. Let us then add to G the edge $\{u, v\}$ with weight w. Under what circumstances will T be a minimal spanning tree of the new graph?

Euler circuit

9.5.6**: An *Euler circuit* for an undirected graph G is a path that starts and ends at the same node and contains each edge of G exactly once.

a) Show that a connected, undirected graph has an Euler circuit if and only if each node is of even degree.

b) Let G be an undirected graph with m edges in which every node is of even degree. Give an $O(m)$ algorithm to construct an Euler circuit for G.

9.6 Depth-First Search

We shall now describe a graph-exploration method that is useful for directed graphs. In Section 5.4 we discussed the preorder and postorder traversals of trees, where we start at the root and recursively explore the children of each node we visit. We can apply almost the same idea to any directed graph.[6] From any node, we recursively explore its successors.

However, we must be careful if the graph has cycles. If there is a cycle, we can wind up calling the exploration function recursively around the cycle forever. For instance, consider the graph of Fig. 9.26. Starting at node a, we might decide to explore node b next. From b we might explore c first, and from c we could explore b first. That gets us into an infinite recursion, where we alternate exploring from b and c. In fact, it doesn't matter in what order we choose to explore successors of b and c. Either we shall get caught in some other cycle, or we eventually explore c from b and explore b from c, infinitely.

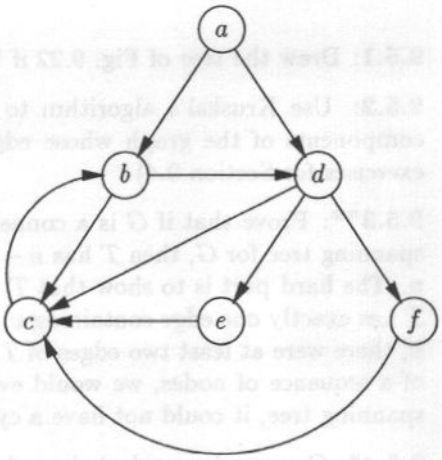


Fig. 9.26. An example directed graph.

There is a simple solution to our problem: We mark nodes as we visit them, and never revisit marked nodes. Then, any node we can reach from our starting node will be reached, but no previously visited node will be revisited. We shall see

[6] Notice that a tree can be thought of as a special case of a directed graph, if we regard the arcs of the tree as directed from parent to child. In fact, a tree is always an acyclic graph as well.

that the time taken by this exploration takes time proportional to the number of arcs explored.

The search algorithm is called *depth-first search* because we find ourselves going as far from the initial node (as "deep") as fast as we can. It can be implemented with a simple data structure. Again, let us assume that the type `NODE` is used to name nodes and that this type is `int`. We represent arcs by adjacency lists. Since we need a "mark" for each node, which can take on the values `VISITED` and `UNVISITED`, we shall create an array of structures to represent the graph. These structures will contain both the mark and the header for the adjacency list.

```
enum MARKTYPE {VISITED, UNVISITED};
typedef struct {
    enum MARKTYPE mark;
    LIST successors;
} GRAPH[MAX];
```

where `LIST` is an adjacency list, defined in the customary manner:

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName
    LIST next;
};
```

We begin by marking all the nodes `UNVISITED`. Recursive function `dfs(u, G)` of Fig. 9.27 works on a node u of some externally defined graph G of type `GRAPH`.

At line (1) we mark u `VISITED`, so we don't call `dfs` on it again. Line (2) initializes p to point to the first cell on the adjacency list for node u. The loop of lines (3) through (7) takes p down the adjacency list, considering each successor, v, of u, in turn.

```
    void dfs(NODE u, GRAPH G)
    {
        LIST p; /* runs down the adjacency list of u */
        NODE v; /* the node in the cell pointed to by p */

(1)     G[u].mark = VISITED;
(2)     p = G[u].successors;
(3)     while (p != NULL) {
(4)         v = p->nodeName;
(5)         if (G[v].mark == UNVISITED)
(6)             dfs(v, G);
(7)         p = p->next;
        }
    }
```

Fig. 9.27. The recursive depth-first search function.

Line (4) sets v to be the "current" successor of u. At line (5) we test whether v has ever been visited before. If so, we skip the recursive call at line (6) and we move p to the next cell of the adjacency list at line (7). However, if v has never been

visited, we start a depth-first search from node v, at line (6). Eventually, we finish the call to `dfs(v, G)`. Then, we execute line (7) to move p down u's adjacency list and go around the loop.

✦ **Example 9.18.** Suppose G is the graph of Fig. 9.26, and, for specificity, assume the nodes on each adjacency list are ordered alphabetically. Initially, all nodes are marked `UNVISITED`. Let us call `dfs(a)`.[7] Node a is marked `VISITED` at line (1), and at line (2) we initialize p to point to the first cell on a's adjacency list. At line (4) v is set to b, since b is the node in the first cell. Since b is currently unvisited, the test of line (5) succeeds, and at line (6) we call `dfs(b)`.

Now, we start a new call to `dfs`, with $u = b$, while the old call with $u = a$ is dormant but still alive. We begin at line (1), marking b `VISITED`. Since c is the first node on b's adjacency list, c becomes the value of v at line (4). Node c is unvisited, so that we succeed at line (5) and at line (6) we call `dfs(c)`.

A third call to `dfs` is now alive, and to begin `dfs(c)`, we mark c `VISITED` and set v to b at line (4), since b is the first, and only, node on c's adjacency list. However, b was already marked `VISITED` at line (1) of the call to `dfs(b)`, so that we skip line (6) and move p down c's adjacency list at line (7). Since c has no more successors, p becomes `NULL`, so that the test of line (3) fails, and `dfs(c)` is finished.

We now return to the call `dfs(b)`. Pointer p is advanced at line (7), and it now points to the second cell of b's adjacency list, which holds node d. We set v to d at line (4), and since d is unvisited, we call `dfs(d)` at line (6).

For the execution of `dfs(d)`, we mark d `VISITED`. Then v is first set to c. But c is visited, and so next time around the loop, $v = e$. That leads to the call `dfs(e)`. Node e has only c as a successor, and so after marking e `VISITED`, `dfs(e)` returns to `dfs(d)`. We next set $v = f$ at line (4) of `dfs(d)`, and call `dfs(f)`. After marking f `VISITED`, we find that f also has only c as a successor, and c is visited.

We are now finished with `dfs(f)`. Since f is the last successor of d, we are also finished with `dfs(d)`, and since d is the last successor of b, we are done with `dfs(b)` as well. That takes us back to `dfs(a)`. Node a has another successor, d, but that node is visited, and so we are done with `dfs(a)` as well.

Figure 9.28 summarizes the action of `dfs` on the graph of Fig. 9.26. We show the stack of calls to `dfs`, with the currently active call at the right. We also indicate the action taken at each step, and we show the value of the local variable v associated with each currently live call, or show that `p = NULL`, indicating that there is no active value for v. ✦

Constructing a Depth-First Search Tree

Because we mark nodes to avoid visiting them twice, the graph behaves like a tree as we explore it. In fact, we can draw a tree whose parent-child edges are some of the arcs of the graph G being searched. If we are in `dfs(u)`, and a call to `dfs(v)` results, then we make v a child of u in the tree. The children of u appear, from left to right, in the order in which *dfs* was called on these children. The node upon which the initial call to *dfs* was made is the root. No node can have *dfs* called on it twice, since it is marked `VISITED` at the first call. Thus, the structure defined is truly a tree. We call the tree a *depth-first search tree* for the given graph.

[7] In what follows, we shall omit the second argument of `dfs`, which is always the graph G.

dfs(a) $v = b$				Call **dfs(b)**
dfs(a) $v = b$	**dfs(b)** $v = c$			Call **dfs(c)**
dfs(a) $v = b$	**dfs(b)** $v = c$	**dfs(c)** $v = b$		Skip; b already visited
dfs(a) $v = b$	**dfs(b)** $v = c$	**dfs(c)** $p =$**NULL**		Return
dfs(a) $v = b$	**dfs(b)** $v = d$			Call **dfs(d)**
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = c$		Skip; c already visited
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = e$		Call **dfs(e)**
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = c$	**dfs(e)** $v = c$	Skip; c already visited
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = e$	**dfs(e)** $p =$**NULL**	Return
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = f$		Call **dfs(f)**
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = f$	**dfs(f)** $v = c$	Skip; c already visited
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = f$	**dfs(f)** $p =$**NULL**	Return
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $p =$**NULL**		Return
dfs(a) $v = b$	**dfs(b)** $p =$**NULL**			Return
dfs(a) $v = d$				Skip; d already visited
dfs(a) $p =$**NULL**				Return

Fig. 9.28. Trace of calls made during depth-first search.

✦ **Example 9.19.** The tree for the exploration of the graph in Fig. 9.26 that was summarized in Fig. 9.28 is seen in Fig. 9.29. We show the *tree arcs*, representing the parent-child relationship, as solid lines. Other arcs of the graph are shown as dotted arrows. For the moment, we should ignore the numbers labeling the nodes. ✦

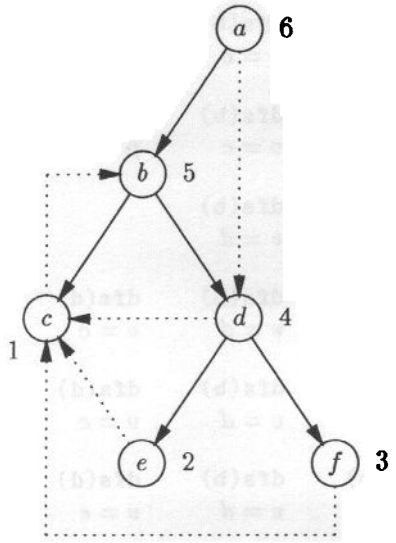


Fig. 9.29. One possible depth-first search tree for the graph of Fig. 9.26.

Classification of Arcs for a Depth-First Search Tree

When we build a depth-first search tree for a graph G, we can classify the arcs of G into four groups. It should be understood that this classification is with respect to a particular depth-first search tree, or equivalently, with respect to the particular order for the nodes in each adjacency list that led to a particular exploration of G. The four kinds of arcs are

1. *Tree arcs*, which are the arcs $u \to v$ such that `dfs(v)` is called by `dfs(u)`.
2. *Forward arcs*, which are arcs $u \to v$ such that v is a proper descendant of u, but not a child of u. For instance, in Fig. 9.29, the arc $a \to d$ is the only forward arc. No tree arc is a forward arc.
3. *Backward arcs*, which are arcs $u \to v$ such that v is an ancestor of u in the tree ($u = v$ is permitted). Arc $c \to b$ is the only example of a backward arc in Fig. 9.29. Any loop, an arc from a node to itself, is classified as backward.
4. *Cross arcs*, which are arcs $u \to v$ such that v is neither an ancestor nor descendant of u. There are three cross arcs in Fig. 9.29: $d \to c$, $e \to c$, and $f \to c$.

Cross arcs go from right to left

In Fig. 9.29, each of the cross arcs go from right to left. It is no coincidence that they do so. Suppose we had in some depth-first search tree a cross arc $u \to v$ such that u was to the left of v. Consider what happens during the call to `dfs(u)`. By the time we finish `dfs(u)`, we shall have considered the arc from u to v. If v has not yet been placed in the tree, then it becomes a child of u in the tree. Since

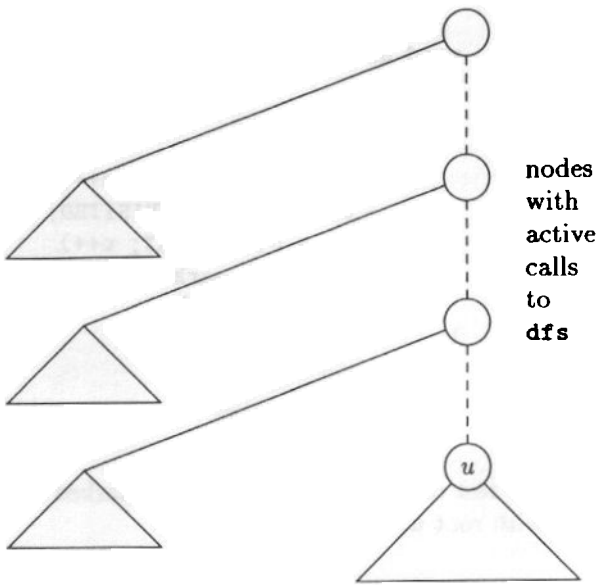


Fig. 9.30. Part of the tree that is built when arc $u \rightarrow v$ is considered.

that evidently did not happen (for then v would not be to the right of u), it must be that v is already in the tree when the arc $u \rightarrow v$ is considered.

However, Fig. 9.30 shows the parts of the tree that exist while `dfs(u)` is active. Since children are added in left-to-right order, no proper ancestor of node u as yet has a child to the right of u. Thus, v can only be an ancestor of u, a descendant of u, or somewhere to the left of u. Thus, if $u \rightarrow v$ is a cross edge, v must be to the left of u, not the right of u as we initially supposed.

The Depth-First Search Forest

We were quite fortunate in Example 9.19 that when we started at node a, we were able to reach all the nodes of the graph of Fig. 9.26. Had we started at any other node, we would not have reached a, and a would not appear in the tree. Thus, the general method of exploring a graph is to construct a sequence of trees. We start at some node u and call `dfs(u)`. If there are nodes not yet visited, we pick one, say v, and call `dfs(v)`. We repeat this process as long as there are nodes not yet assigned to any tree.

When all nodes have been assigned a tree, we list the trees, from left to right, in the order of their construction. This list of trees is called the *depth-first search forest*. In terms of the data types `NODE` and `GRAPH` defined earlier, we can explore an entire externally defined graph G, starting the search on as many roots as necessary by the function of Fig. 9.31. There, we assume that the type `NODE` is `int`, and MAX is the number of nodes in G.

In lines (1) and (2) we initialize all nodes to be `UNVISITED`. Then, in the loop of lines (3) to (5), we consider each node u in turn. When we consider u, if that node has not yet been added to any tree, it will still be marked unvisited when we make the test of line (4). In that case, we call `dfs(u, G)` at line (5) and explore the depth-first search tree with root u. In particular, the first node always becomes the root of a tree. However, if u has already been added to a tree when we perform

```
    void dfsForest(GRAPH G);
    {
        NODE u;

(1)     for (u = 0; u < MAX; u++)
(2)         G[u].mark = UNVISITED;
(3)     for (u = 0; u < MAX; u++)
(4)         if (G[u].mark == UNVISITED)
(5)             dfs(u, G);
    }
```

Fig. 9.31. Exploring a graph by exploring as many trees as necessary.

the test of line (4), then u will be marked `VISITED`, and so we do not create a tree with root u.

✦ **Example 9.20.** Suppose we apply the above algorithm to the graph of Fig. 9.26, but let d be the node whose name is 0; that is, d is the first root of a tree for the depth-first spanning forest. We call `dfs(d)`, which constructs the first tree of Fig. 9.32. Now, all nodes but a are visited. As u becomes each of the various nodes in the loop of lines (3) to (5) of Fig. 9.31, the test of line (4) fails except when $u = a$. Then, we create the one-node second tree of Fig. 9.32. Note that both successors of a are marked `VISITED` when we call `dfs(a)`, and so we do not make any recursive calls from `dfs(a)`. ✦

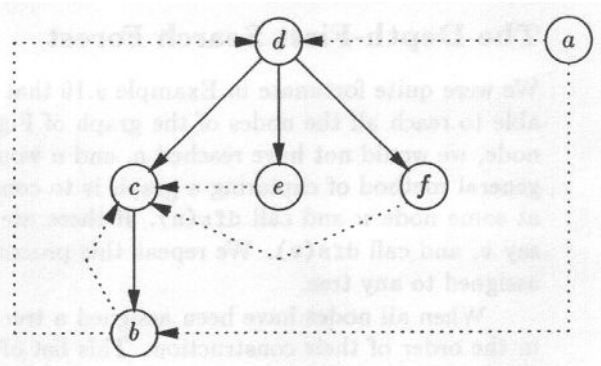


Fig. 9.32. A depth-first search forest.

When we present the nodes of a graph as a depth-first search forest, the notions of forward, backward, and tree arcs apply as before. However, the notion of a cross arc must be extended to include arcs that run from one tree to a tree to its left. Examples of such cross arcs are $a \to b$ and $a \to d$ in Fig. 9.32.

The rule that cross arcs always go from right to left continues to hold. The reason is also the same. If there were a cross arc $u \to v$ that went from one tree to a tree to the right, then consider what happens when we call `dfs(u)`. Since v

The Perfection of Depth-First Search

Regardless of the relationship between the numbers of nodes and arcs, the running time of the depth-first exploration of a graph takes time proportional to the "size" of the graph, that is, the sum of the numbers of nodes and arcs. Thus, depth-first search is, to within a constant factor, as fast as any algorithm that "looks at" the graph.

was not added to the tree being formed at the moment, it must already have been in some tree. But the trees to the right of u have not yet been created, and so v cannot be part of one of them.

Running Time of the Depth-First Search Algorithm

Let G be a graph with n nodes and let m be the larger of the number of nodes and the number of arcs. Then `dfsForest` of Fig. 9.31 takes $O(m)$ time. The proof of this fact requires a trick. When calculating the time taken by a call `dfs(u)`, we shall not count the time taken by any recursive calls to `dfs` at line (6) in Fig. 9.27, as Section 3.9 suggested we should. Rather, observe that we call `dfs(u)` once for each value of u. Thus, if we sum the cost of each call, exclusive of its recursive calls, we get the total time spent in all the calls as a group.

Notice that the while-loop of lines (3) to (7) in Fig. 9.27 can take a variable amount of time, even excluding the time spent in recursive calls to `dfs`, because the number of successors of node u could be any number from 0 to n. Suppose we let m_u be the out-degree of node u, that is, the number of successors of u. Then the number of times around the while-loop during the execution of `dfs(u)` is surely m_u. We do not count the execution of `dfs(v, G)` at line (6) when assessing the running time of `dfs(u)`, and the body of the loop, exclusive of this call, takes $O(1)$ time. Thus, the total time spent in the loop of lines (3) to (7), exclusive of time spent in recursive calls is $O(1 + m_u)$; the additional 1 is needed because m_u might be 0, in which case we still take $O(1)$ time for the test of (3). Since lines (1) and (2) of `dfs` take $O(1)$ time, we conclude that, neglecting recursive calls, `dfs(u)` takes time $O(1 + m_u)$ to complete.

Now we observe that during the running of `dfsForest`, we call `dfs(u)` exactly once for each value of u. Thus, the total time spent in all these calls is big-oh of the sum of the times spent in each, that is, $O(\sum_u (1 + m_u))$. But $\sum_u m_u$ is just the number of arcs in the graph, that is, at most m,[8] since each arc emanates from some one node. The number of nodes is n, so that $\sum_u 1$ is just n. As $n \leq m$, the time taken by all the calls to `dfs` is thus $O(m)$.

Finally, we must consider the time taken by `dfsForest`. This program, in Fig. 9.31, consists of two loops, each iterated n times. The bodies of the loops are easily seen to take $O(1)$ time, exclusive of the calls to `dfs`, and so the cost of the loops is $O(n)$. This time is dominated by the $O(m)$ time of the calls to `dfs`. Since the time for the `dfs` calls is already accounted for, we conclude that `dfsForest`, together with all its calls to `dfs`, takes $O(m)$ time.

[8] In fact, the sum of the m_u's will be exactly m, except in the case that the number of nodes exceeds the number of arcs; recall that m is the larger of the numbers of nodes and arcs.

Postorder Traversals of Directed Graphs

Once we have a depth-first search tree, we could number its nodes in postorder. However, there is an easy way to do the numbering during the search itself. We simply attach the number to a node u as the last thing we do before `dfs(u)` completes. Then, a node is numbered right after all its children are numbered, just as in a postorder numbering.

```
    int k; /* counts visited nodes */

    void dfs(NODE u, GRAPH G)
    {
        LIST p; /* points to cells of adjacency list of u */
        NODE v; /* the node in the cell pointed to by p */
(1)     G[u].mark = VISITED;
(2)     p = G[u].successors;
(3)     while (p != NULL) {
(4)         v = p->nodeName;
(5)         if (G[v].mark == UNVISITED)
(6)             dfs(v, G);
(7)         p = p->next;
        }
(8)     ++k;
(9)     G[u].postorder = k;
    }

    void dfsForest(GRAPH G)
    {
        NODE u;
(10)    k = 0;
(11)    for (u = 0; u < MAX; u++)
(12)        G[u].mark = UNVISITED;
(13)    for (u = 0; u < MAX; u++)
(14)        if (G[u].mark == UNVISITED)
(15)            dfs(u, G);
    }
```

Fig. 9.33. Procedure to number the nodes of a directed graph in postorder.

✦ **Example 9.21.** The tree of Fig. 9.29, which we constructed by depth-first search of the graph in Fig. 9.26, has the postorder numbers labeling the nodes. If we examine the trace of Fig. 9.28, we see that the first call to return is `dfs(c)`, and node c is given the number 1. Then, we visit d, then e, and return from the call to e. Therefore, e's number is 2. Similarly, we visit and return from f, which is numbered 3. At that point, we have completed the call on d, which gets number 4. That completes the call to `dfs(b)`, and the number of b is 5. Finally, the original

call to a returns, giving a the number 6. Notice that this order is exactly the one we would get if we simply walked the tree in postorder. ✦

We can assign the postorder numbers to the nodes with a few simple modifications to the depth-first search algorithm we have written so far; these changes are summarized in Fig. 9.33.

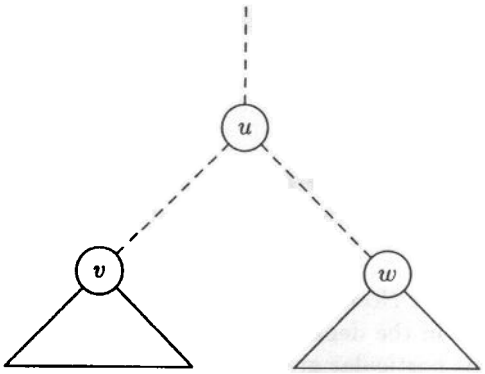
1. In the `GRAPH` type, we need an additional field for each node, called `postorder`. For the graph G, we place the postorder number of node u in `G[u].postorder`. This assignment is accomplished at line (9) of Fig. 9.33.

2. We use a global variable `k` to count nodes in postorder. This variable is defined externally to `dfs` and `dfsForest`. As seen in Fig. 9.33, we initialize k to 0 in line (10) of `dfsForest`, and just before assigning a postorder number, we increment k by 1 at line (8) in `dfs`.

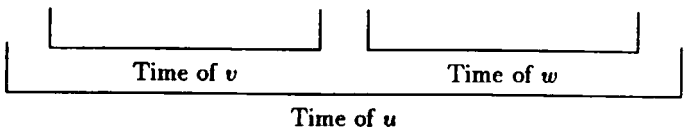
Notice that as a result, when there is more than one tree in the depth-first search forest, the first tree gets the lowest numbers, the next tree gets the next numbers in order, and so on. For example, in Fig. 9.32, a would get the postorder number 6.

Special Properties of Postorder Numbers

The impossibility of cross arcs that go left to right tells us something interesting and useful about the postorder numbers and the four types of arcs in a depth-first presentation of a graph. In Fig. 9.34(a) we see three nodes, u, v, and w, in a depth-first presentation of a graph. Nodes v and w are descendants of u, and w is to the right of v. Figure 9.34(b) shows the duration of activity for the calls to `dfs` for each of these nodes.



(a) Three nodes in a depth-first tree.



(b) Active intervals for their calls to `dfs`.

Fig. 9.34. Relationship between position in tree and duration of calls.

We can make several observations. First, the call to `dfs` on a descendant like v is active for only a subinterval of the time during which the call on an ancestor, like u, is active. In particular, the call to `dfs(v)` terminates before the call to `dfs(u)` does. Thus, the postorder number of v must be less than the postorder number of u whenever v is a proper descendant of u.

Second, if w is to the right of v, then the call to `dfs(w)` cannot begin until after the call to `dfs(v)` terminates. Thus, whenever v is to the left of w, the postorder number of v is less than that of w. Although not shown in Fig. 9.34, the same is true even if v and w are in different trees of the depth-first search forest, with v's tree to the left of w's tree.

We can now consider the relationship between the postorder numbers of u and v for each arc $u \to v$.

1. If $u \to v$ is a tree arc or forward arc, then v is a descendant of u, and so v precedes u in postorder.

2. If $u \to v$ is a cross arc, then we know v is to the left of u, and again v precedes u in postorder.

3. If $u \to v$ is a backward arc and $v \neq u$, then v is a proper ancestor of u, and so v follows u in postorder. However, $v = u$ is possible for a backward arc, since a loop is a backward arc. Thus, in general, for backward arc $u \to v$, we know that the postorder number of v is at least as high as the postorder number of u.

In summary, we see that in postorder, the head of an arc precedes the tail, unless the arc is a backward arc; in which case the tail precedes or equals the head. Thus, we can identify the backward arcs simply by finding those arcs whose tails are equal to or less than their heads in postorder. We shall see a number of applications of this idea in the next section.

EXERCISES

9.6.1: For the tree of Fig. 9.5 (see the exercises for Section 9.2), give two depth-first search trees starting with node a. Give a depth-first search tree starting with node d.

9.6.2*: No matter which node we start with in Fig. 9.5, we wind up with only one tree in the depth-first search forest. Explain briefly why that must be the case for this particular graph.

9.6.3: For each of your trees of Exercise 9.6.1, indicate which of the arcs are tree, forward, backward, and cross arcs.

9.6.4: For each of your trees of Exercise 9.6.1, give the postorder numbers for the nodes.

9.6.5*: Consider the graph with three nodes, a, b, and c, and the two arcs $a \to b$ and $b \to c$. Give all the possible depth-first search forests for this graph, considering all possible starting nodes for each tree. What is the postorder numbering of the nodes for each forest? Are the postorder numbers always the same for this graph?

9.6.6*: Consider the generalization of the graph of Exercise 9.6.5 to a graph with n nodes, $a_1, a_2, \ldots, a_n$, and arcs $a_1 \rightarrow a_2$, $a_2 \rightarrow a_3, \ldots, a_{n-1} \rightarrow a_n$. Prove by complete induction on n that this graph has 2^{n-1} different depth-first search forests. *Hint*: It helps to remember that $1 + 1 + 2 + 4 + 8 + \cdots + 2^i = 2^{i+1}$, for $i \geq 0$.

9.6.7*: Suppose we start with a graph G and add a new node x that is a predecessor of all other nodes in G. If we run `dfsForest` of Fig. 9.31 on the new graph, starting at node x, then a single tree results. If we then delete x from this tree, several trees may result. How do these trees relate to the depth-first search forest of the original graph G?

9.6.8**: Suppose we have a directed graph G, from whose representation we have just constructed a depth-first spanning forest F by the algorithm of Fig. 9.31. Let us now add the arc $u \rightarrow v$ to G to form a new graph H, whose representation is exactly that of G, except that node v now appears somewhere on the adjacency list for node u. If we now run Fig. 9.31 on this representation of H, under what circumstances will the same depth-first forest F be constructed? That is, when will the tree arcs for H are exactly the same as the tree arcs for G?

9.7 Some Uses of Depth-First Search

In this section, we see how depth-first search can be used to solve some problems quickly. As previously, we shall here use n to represent the number of nodes of a graph, and we shall use m for the larger of the number of nodes and the number of arcs; in particular, we assume that $n \leq m$ is always true. Each of the algorithms presented takes $O(m)$ time, on a graph represented by adjacency lists. The first algorithm determines whether a directed graph is acyclic. Then for those graphs that are acyclic, we see how to find a topological sort of the nodes (topological sorting was discussed in Section 7.10; we shall review the definitions at the appropriate time). We also show how to compute the transitive closure of a graph (see Section 7.10 again), and how to find connected components of an undirected graph faster than the algorithm given in Section 9.4.

Finding Cycles in a Directed Graph

During a depth-first search of a directed graph G, we can assign a postorder number to all the nodes in $O(m)$ time. Recall from the last section that we discovered the only arcs whose tails are equal to or less than their heads in postorder are the backward arcs. Whenever there is a backward arc, $u \rightarrow v$, in which the postorder number of v is at least as large as the postorder number of u, there must be a cycle in the graph, as suggested by Fig. 9.35. The cycle consists of the arc from u to v, and the path in the tree from v to its descendant u.

The converse is also true; that is, if there is a cycle, then there must be a backward arc. To see why, suppose there is a cycle, say $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_1$, and let the postorder number of node v_i be p_i, for $i = 1, 2, \ldots, k$. If $k = 1$, that is, the cycle is a single arc, then surely $v_1 \rightarrow v_1$ is a backward arc in any depth-first presentation of G.

If $k > 1$, suppose that none of the arcs $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3$, and so on, up to $v_{k-1} \rightarrow v_k$ are backward. Then each head precedes each tail in postorder, and so the postorder numbers $p_1, p_2, \ldots, p_k$ form a decreasing sequence. In particular,

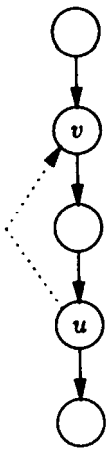


Fig. 9.35. Every backward arc forms a cycle with tree arcs.

$p_k < p_1$. Then consider the arc $v_k \rightarrow v_1$ that completes the cycle. The postorder number of its tail, which is p_k, is less than the postorder number of its head, p_1, and so this arc is a backward arc. That proves there must be some backward arc in any cycle.

As a result, after computing the postorder numbers of all nodes, we simply examine all the arcs, to see if any has a tail less than or equal to its head, in postorder. If so, we have found a backward arc, and the graph is cyclic. If there is no such arc, the graph is acyclic. Figure 9.36 shows a function that tests whether an externally defined graph G is acyclic, using the data structure for graphs described in the previous section. It also makes use of the function **dfsForest** defined in Fig. 9.33 to compute the postorder numbers of the nodes of G.

```
    BOOLEAN testAcyclic(GRAPH G)
    {
        NODE u, v; /* u runs through all the nodes */
        LIST p; /* p points to each cell on the adjacency list
                   for u; v is a node on the adjacency list */

(1)     dfsForest(G);
(2)     for (u = 0; u < MAX; u++) {
(3)         p = G[u].successors;
(4)         while (p != NULL) {
(5)             v = p->nodeName;
(6)             if (G[u].postorder <= G[v].postorder)
(7)                 return FALSE;
(8)             p = p->next;
            }
        }
(9)     return TRUE;
    }
```

Fig. 9.36. Function to determine whether a graph G is acyclic.

After calling **dfsForest** to compute postorder numbers at line (1), we examine each node u in the loop of lines (2) through (8). Pointer p goes down the adjacency list for u, and at line (5), v in turn becomes each successor of u. If at line (6) we find that u equals or precedes v in postorder, then we have found a backward arc $u \rightarrow v$, and we return **FALSE** at line (7). If we find no such backward arc, we return **TRUE** at line (9).

Running Time of the Acyclicity Test

As before, let n be the number of nodes of graph G and let m be the larger of the number of nodes and the number of arcs. We already know that the call to **dfsForest** at line (1) of Fig. 9.36 takes $O(m)$ time. Lines (5) to (8), the body of the while-loop, evidently take $O(1)$ time. To get a good bound on the time for the while-loop itself, we must use the trick that was used in the previous section to bound the time of depth-first search. Let m_u be the out-degree of node u. Then we go around the loop of lines (4) to (8) m_u times. Thus, the time spent in lines (4) to (8) is $O(1 + m_u)$.

Line (3) only takes $O(1)$ time, and so the time spent in the for-loop of lines (2) to (8) is $O(\sum_u (1 + m_u))$. As observed in the previous section, the sum of 1 is $O(n)$, and the sum of m_u is m. Since $n \leq m$, the time for the loop of lines (2) to (8) is $O(m)$. That is the same as the time for line (1), and line (9) takes $O(1)$ time. Thus, the entire acyclicity test takes $O(m)$ time. As for depth-first search itself, the time to detect cycles is, to within a constant factor, just the time it takes to look at the entire graph.

Topological Sorting

Suppose we know that a directed graph G is acyclic. As for any graph, we may find a depth-first search forest for G and thereby determine a postorder for the nodes of G. Suppose $(v_1, v_2, \ldots, v_n)$ is a list of the nodes of G in the reverse of postorder; that is, v_1 is the node numbered n in postorder, v_2 is numbered $n-1$, and in general, v_i is the node numbered $n - i + 1$ in postorder.

The order of the nodes on this list has the property that all arcs of G go forward in the order. To see why, suppose $v_i \rightarrow v_j$ is an arc of G. Since G is acyclic, there are no backward arcs. Thus, for every arc, the head precedes the tail. That is, v_j precedes v_i in postorder. But the list is the reverse of postorder, and so v_i precedes v_j on the list. That is, every tail precedes the corresponding head in the list order.

Topological order

An order for the nodes of a graph G with the property that for every arc of G the tail precedes the head is called a *topological order*, and the process of finding such an order for the nodes is called *topological sorting*. Only acyclic graphs have a topological order, and as we have just seen, we can produce a topological order for an acyclic graph $O(m)$ time, where m is the larger of the number of nodes and arcs, by performing a depth-first search. As we are about to give a node its postorder number, that is, as we complete the call to **dfs** on that node, we push the node onto a stack. When we are done, the stack is a list in which the nodes appear in postorder, with the highest at the top (front). That is the reverse postorder we desire. Since the depth-first search takes $O(m)$ time, and pushing the nodes onto a stack takes only $O(n)$ time, the whole process takes $O(m)$ time.

Applications of Topological Order and Cycle Finding

There are a number of situations in which the algorithms discussed in this section will prove useful. Topological ordering comes in handy when there are constraints on the order in which we do certain tasks, which we represent by nodes. If we draw an arc from u to v whenever we must do task u before v, then a topological order is an order in which we can perform all the tasks. An example in Section 7.10 about putting on shoes and socks illustrated this type of problem.

A similar example is the calling graph of a nonrecursive collection of functions, in which we wish to analyze each function after we have analyzed the functions it calls. As the arcs go from caller to called function, the reverse of a topological order, that is, the postorder itself, is an order in which we can analyze the function, making sure that we only work on a function after we have worked on all the functions it calls.

In other situations, it is sufficient to run the cycle test. For example, a cycle in the graph of task priorities tells us there is no order in which all the tasks can be done, and a cycle in a calling graph tells us there is recursion.

✦ **Example 9.22.** In Fig. 9.37(a) is an acyclic graph, and in Fig. 9.37(b) is the depth-first search forest we get by considering the nodes in alphabetic order. We also show in Fig. 9.37(b) the postorder numbers that we get from this depth-first search. If we list the nodes highest postorder number first, we get the topological order (d, e, c, f, b, a). The reader should check that each of the eight arcs in Fig. 9.37(a) has a tail that precedes its head according to this list. There are, incidentally, three other topological orders for this graph, such as (d, c, e, b, f, a). ✦

The Reachability Problem

Reachable set

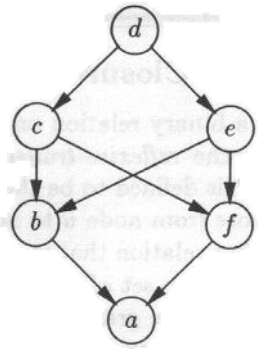
A natural question to ask about a directed graph is, given a node u, which nodes can we reach from u by following arcs? We call this set of nodes the *reachable set* for node u. In fact, if we ask this *reachability* question for each node u, then we know for which pairs of nodes (u, v) there is a path from u to v.

The algorithm for solving reachability is simple. If we are interested in node u, we mark all nodes `UNVISITED` and call `dfs(u)`. We then examine all the nodes again. Those marked `VISITED` are reachable from u, and the others are not. If we then wish to find the nodes reachable from another node v, we set all the nodes to `UNVISITED` again and call `dfs(v)`. This process may be repeated for as many nodes as we like.

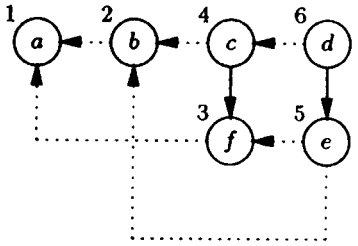
✦ **Example 9.23.** Consider the graph of Fig. 9.37(a). If we start our depth-first search from node a, we can go nowhere, since there are no arcs out of a. Thus, `dfs(a)` terminates immediately. Since only a is visited, we conclude that a is the only node reachable from a.

If we start with b, we can reach a, but that is all; the reachable set for b is $\{a, b\}$. Similarly, from c we reach $\{a, b, c, f\}$, from d we reach all the nodes, from e we reach $\{a, b, e, f\}$, and from f we can reach only $\{a, f\}$.

For another example, consider the graph of Fig. 9.26. From a we can reach all the nodes. From any node but a, we can reach all the nodes except a. ✦



(a) A directed acyclic graph.



(b) A depth-first search forest.

Fig. 9.37. Topologically sorting an acyclic graph.

Running Time of the Reachability Test

Let us assume we have a directed graph G with n nodes and m arcs. We shall also assume G is represented by the data type `GRAPH` of the previous section. First, suppose we want to find the reachable set for a node u. Initializing the nodes to be `UNVISITED` takes $O(n)$ time. The call to `dfs(u, G)` takes $O(m)$ time, and examining the nodes again to see which are visited takes $O(n)$ time. While we examine the nodes, we could also create a list of those nodes that are reachable from u, still taking only $O(n)$ time. Thus, finding the reachable set for one node takes $O(m)$ time.

Now suppose we want the reachable sets for all n nodes. We may repeat the algorithm n times, once for each node. Thus, the total time is $O(nm)$.

Finding Connected Components by Depth-First Search

In Section 9.4, we gave an algorithm for finding the connected components of an undirected graph with n nodes, and with m equal to the larger of the number of nodes and edges, in $O(m \log n)$ time. The tree structure we used to merge components is of interest in its own right; for example, we used it to help implement Kruskal's minimal spanning tree algorithm. However, we can find connected components more efficiently if we use depth-first search.
suffices.

The idea is to treat the undirected graph as if it were a directed graph with each edge replaced by arcs in both directions. If we represent the graph by adjacency lists, then we do not even have to make any change to the representation. Now

Transitive Closure and Reflexive-Transitive Closure

Let R be a binary relation on a set S. The reachability problem can be viewed as computing the *reflexive-transitive closure* of R, which is usually denoted R^*. The relation R^* is defined to be the set of pairs (u, v) such that there is a path of length zero or more from node u to node v in the graph represented by R.

Another relation that is very similar is R^+, the *transitive closure* of R, which is defined to be the set of pairs (u, v) such that there is a path of length one or more from u to v in the graph represented by R. The distinction between R^* and R^+ is that (u, u) is always in R^* for every u in S, whereas (u, u) is in R^+ if and only if there is a cycle of length one or more from u to u. To compute R^+ from R^*, we just have to check whether or not each node u has some entering arc from one of its reachable nodes, including itself; if it does not, we remove u from its own reachable set.

construct the depth-first search forest for the directed graph. Each tree in the forest is one connected component of the undirected graph.

To see why, first note that the presence of an arc $u \to v$ in the directed graph indicates that there is an edge $\{u, v\}$. Thus, all the nodes of a tree are connected.

Now we must show the converse, that if two nodes are connected, then they are in the same tree. Suppose there were a path in the undirected graph between two nodes u and v that are in different trees. Say the tree of u was constructed first. Then there is a path in the directed graph from u to v, which tells us that v, and all the nodes on this path, should have been added to the tree with u. Thus, nodes are connected in the undirected graph if and only if they are in the same tree; that is, the trees are the connected components.

✦ **Example 9.24.** Consider the undirected graph of Fig. 9.4 again. One possible depth-first search forest we might construct for this graph is shown in Fig. 9.38. Notice how the three depth-first search trees correspond to the three connected components. ✦

EXERCISES

9.7.1: Find all the topological orders for the graph of Fig. 9.37.

9.7.2*: Suppose R is a partial order on domain D. We can represent R by its graph, where the nodes are the elements of D and there is an arc $u \to v$ whenever uRv and $u \neq v$. Let $(v_1, v_2, \ldots, v_n)$ be a topological ordering of the graph of R. Let T be the relation defined by $v_i T v_j$ whenever $i \leq j$. Show that

a) T is a total order, and

b) The pairs in R are a subset of the pairs in T; that is, T is a total order containing the partial order R.

9.7.3: Apply depth-first search to the graph of Fig. 9.21 (after converting it to a symmetric directed graph), to find the connected components.

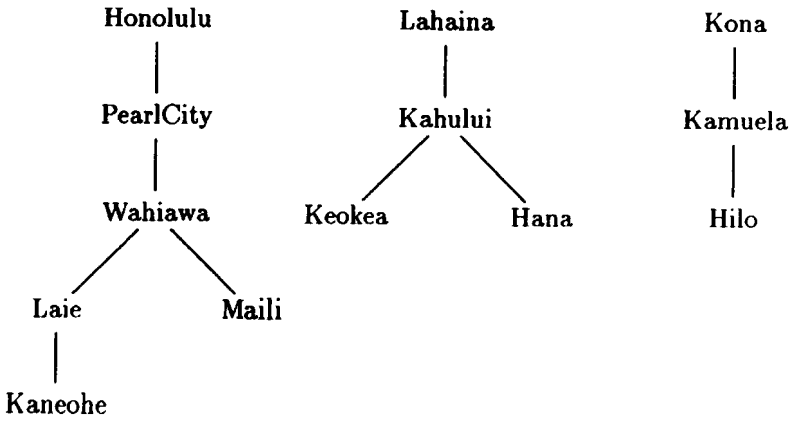


Fig. 9.38. The depth-first search forest divides an undirected graph into connected components.

9.7.4: Consider the graph with arcs $a \to c$, $b \to a$, $b \to c$, $d \to a$, and $e \to c$.

a) Test the graph for cycles.
b) Find all the topological orders for the graph.
c) Find the reachable set of each node.

9.7.5*: In the next section we shall consider the general problem of finding shortest paths from a source node s. That is, we want for each node u the length of the shortest path from s to u if one exists. When we have a directed, acyclic graph, the problem is easier. Give an algorithm that will compute the length of the shortest path from node s to each node u (infinity if no such path exists) in a directed, acyclic graph G. Your algorithm should take $O(m)$ time, where m is the larger of the number of nodes and arcs of G. Prove that your algorithm has this running time. *Hint*: Start with a topological sort of G, and visit each node in turn. On visiting a node u, calculate the shortest distance from s to u in terms of the already calculated shortest distances to the predecessors of u.

9.7.6*: Give algorithms to compute the following for a directed, acyclic graph G. Your algorithms should run in time $O(m)$, where m is the larger of the number of nodes and arcs of G, and you should prove that this running time is all that your algorithm requires. *Hint*: Adapt the idea of Exercise 9.7.5.

a) For each node u, find the length of the longest path from u to anywhere.
b) For each node u, find the length of the longest path to u from anywhere.
c) For a given source node s and for all nodes u of G, find the length of the *longest* path from s to u.
d) For a given source node s and for all nodes u of G, find the length of the longest path from u to s.
e) For each node u, find the length of the longest path through u.

9.8 Dijkstra's Algorithm for Finding Shortest Paths

Suppose we have a graph, which could be either directed or undirected, with labels on the arcs (or edges) to represent the "length" of that arc. An example is Fig. 9.4, which showed the distance along certain roads of the Hawaiian Islands. It is quite common to want to know the minimum distance between two nodes; for example, maps often include tables of driving distance as a guide to how far one can travel in a day, or to help determine which of two routes (that go through different intermediate cities) is shorter. A similar kind of problem would associate with each arc the time it takes to travel along that arc, or perhaps the cost of traveling that arc. Then the minimum "distance" between two nodes would correspond to the traveling time or the fare, respectively.

Minimum distance

In general, the *distance* along a path is the sum of the labels of that path. The *minimum distance from node u to node v* is the minimum of the distance of any path from u to v.

✦ **Example 9.25.** Consider the map of Oahu in Fig. 9.10. Suppose we want to find the minimum distance from Maili to Kaneohe. There are several paths we could choose. One useful observation is that, as long as the labels of the arcs are nonnegative, the minimum-distance path need never have a cycle. For we could skip that cycle and find a path between the same two nodes, but with a distance no greater than that of the path with the cycle. Thus, we need only consider

1. The path through Pearl City and Honolulu.
2. The path through Wahiawa, Pearl City, and Honolulu.
3. The path through Wahiawa and Laie.
4. The path through Pearl City, Wahiawa, and Laie.

The distances of these paths are 44, 51, 67, and 84, respectively. Thus, the minimum distance from Maili to Kaneohe is 44. ✦

Source node

If we wish to find the minimum distance from one given node, called the *source* node, to all the nodes of the graph, one of the most efficient techniques to use is a method called *Dijkstra's algorithm*, the subject of this section. It turns out that if all we want is the distance from one node u to another node v, the best way is to run Dijkstra's algorithm with u as the source node and stop when we deduce the distance to v. If we want to find the minimum distance between every pair of nodes, there is an algorithm that we shall cover in the next section, called Floyd's algorithm, that sometimes is preferable to running Dijkstra's algorithm with every node as a source.

The essence of Dijkstra's algorithm is that we discover the minimum distance from the source to other nodes in the order of those minimum distances, that is, closest nodes first. As Dijkstra's algorithm proceeds, we have a situation like that suggested in Fig. 9.39. In the graph G there are certain nodes that are *settled*, that is, their minimum distance is known; this set always includes s, the source node. For the unsettled nodes v, we record the length of the shortest *special path*, which is a path that starts at the source node, travels only through settled nodes, then at the last step jumps out of the settled region to v.

Settled node

Special path

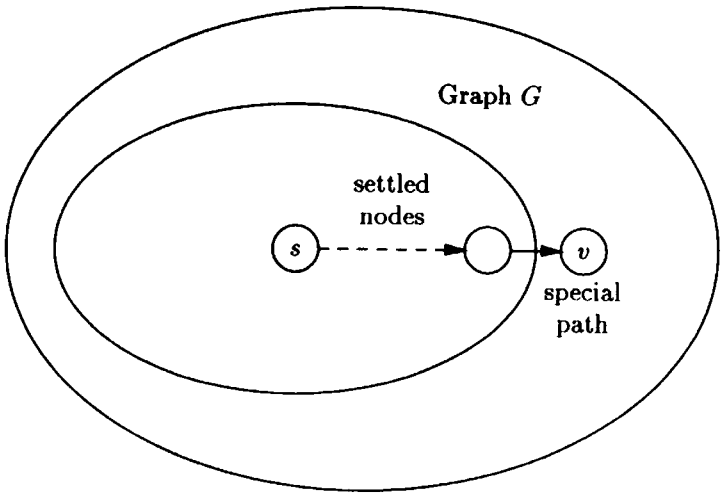


Fig. 9.39. Intermediate stage during the execution of Dijkstra's algorithm.

We maintain a value $dist(u)$ for every node u. If u is a settled node, then $dist(u)$ is the length of the shortest path from the source to u. If u is not settled, then $dist(u)$ is the length of the shortest special path from the source to u. Initially, only the source node s is settled, and $dist(s) = 0$, since the path consisting of s alone surely has distance 0. If there is an arc from s to u, then $dist(u)$ is the label of that arc. Notice that when only s is settled, the only special paths are the arcs out of s, so that $dist(u)$ should be the label of the arc $s \rightarrow u$ if there is one. We shall use a defined constant `INFTY`, that is intended to be larger than the distance along any path in the graph G. `INFTY` serves as an "infinite" value and indicates that no special paths have yet been discovered. That is, initially, if there is no arc $s \rightarrow u$, then $dist(u) =$ `INFTY`.

Now suppose we have some settled and some unsettled nodes, as suggested by Fig. 9.39. We find the node v that is unsettled, but has the smallest $dist$ value of any unsettled node. We "settle" v by

1. Accepting $dist(v)$ as the minimum distance from s to v.
2. Adjusting the value of $dist(u)$, for all nodes u that remain unsettled, to account for the fact that v is now settled.

The adjustment required by step (2) is the following. We compare the old value of $dist(u)$ with the sum of $dist(v)$ and label of the arc $v \rightarrow u$, and if the latter sum is smaller, we replace $dist(u)$ by that sum. If there is no arc $v \rightarrow u$, then we do not adjust $dist(u)$.

✦ **Example 9.26.** Consider the map of Oahu in Fig. 9.10. That graph is undirected, but we shall assume edges are arcs in both directions. Let the source be Honolulu. Then initially, only Honolulu is settled and its distance is 0. We can set $dist$(PearlCity) to 13 and $dist$(Kaneohe) to 11, but other cities, having no arc from Honolulu, are given distance `INFTY`. The situation is shown in the first column of Fig. 9.40. The star on distances indicates that the node is settled.

Among the unsettled nodes, the one with the smallest distance is now Kaneohe,

	ROUND				
CITY	(1)	(2)	(3)	(4)	(5)
Honolulu	0*	0*	0*	0*	0*
PearlCity	13	13	13*	13*	13*
Maili	INFTY	INFTY	33	33	33*
Wahiawa	INFTY	INFTY	25	25*	25*
Laie	INFTY	35	35	35	35
Kaneohe	11	11*	11*	11*	11*

VALUES OF *dist*

Fig. 9.40. Stages in the execution of Dijkstra's algorithm.

and so this node is settled. There are arcs from Kaneohe to Honolulu and Laie. The arc to Honolulu does not help, but the value of *dist*(Kaneohe), which is 11, plus the label of the arc from Kaneohe to Laie, which is 24, totals 35, which is less than "infinity," the current value of *dist*(Laie). Thus, in the second column, we have reduced the distance to Laie to 35. Kaneohe is now settled.

In the next round, the unsettled node with the smallest distance is Pearl City, with a distance of 13. When we make Pearl City settled, we must consider the neighbors of Pearl City, which are Maili and Wahiawa. We reduce the distance to Maili to 33 (the sum of 13 and 20), and we reduce the distance to Wahiawa to 25 (the sum of 13 and 12). The situation is now as in column (3).

Next to be settled is Wahiawa, with a distance of 25, least among the currently unsettled nodes. However, that node does not allow us to reduce the distance to any other node, and so column (4) has the same distances as column (3). Similarly, we next settle Maili, with a distance of 33, but that does not reduce any distances, leaving column (5) the same as column (4). Technically, we have to settle the last node, Laie, but the last node cannot affect any other distances, and so column (5) gives the shortest distances from Honolulu to all six cities. ✦

Why Dijkstra's Algorithm Works

In order to show that Dijkstra's algorithm works, we must assume that the labels of arcs are nonnegative.[9] We shall prove by induction on k that when there are k settled nodes,

a) For each settled node u, *dist*(u) is the minimum distance from s to u, and the shortest path to u consists only of settled nodes.

b) For each unsettled node u, *dist*(u) is the minimum distance of any special path from s to u (**INFTY** if no such path exists).

BASIS. For $k = 1$, s is the only settled node. We initialize *dist*(s) to 0, which satisfies (a). For every other node u, we initialize *dist*(u) to be the label of the arc $s \rightarrow u$ if it exists, and **INFTY** if not. Thus, (b) is satisfied.

[9] When labels are allowed to be negative, we can find graphs for which Dijkstra's algorithm gives incorrect answers.

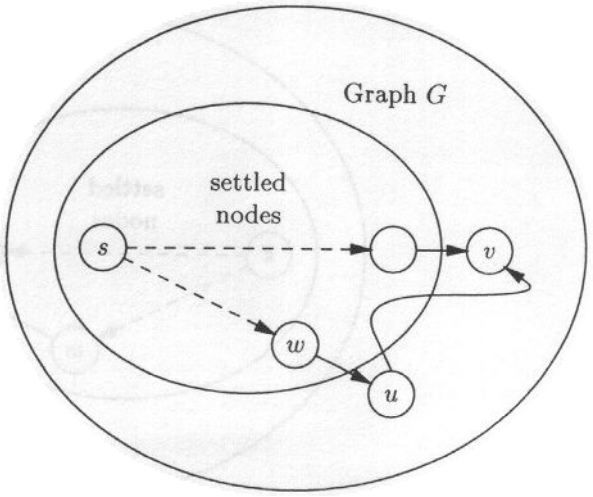


Fig. 9.41. Hypothetical shorter path to v, through w and u.

INDUCTION. Now assume (a) and (b) hold after k nodes have been settled, and let v be the $(k+1)$st node settled. We claim that (a) still holds, because $dist(v)$ is the least distance of any path from s to v. Suppose not. By part (b) of the inductive hypothesis, when k nodes are settled, $dist(v)$ is the minimum distance of any special path to v, and so there must be some shorter nonspecial path to v. As suggested in Fig. 9.41, this path must leave the settled nodes at some node w (which could be s), and go to some unsettled node u. From there, the path could meander in and out of the settled nodes, until it finally arrives at v.

However, v was chosen to be the $(k+1)$st node settled, which means that at this time, $dist(u)$ could not be less than $dist(v)$, or else we would have selected u as the $(k+1)$st node. By (b) of the inductive hypothesis, $dist(u)$ is the minimum length of any special path to u. But the path from s to w to u in Fig. 9.41 is a special path, so that its distance is at least $dist(u)$. Thus, the supposed shorter path from s to v through w and u has a distance that is at least $dist(v)$, because the initial part from s to u already has distance $dist(u)$, and $dist(u) \geq dist(v)$.[10] Thus, (a) holds for $k+1$ nodes, that is, (a) continues to hold when we include v among the settled nodes.

Now we must show that (b) holds when we add v to the settled nodes. Consider some node u that remains unsettled when we add v to the settled nodes. On the shortest special path to u, there must be some penultimate (next-to-last) node; this node could either be v or some other node w. The two possibilities are suggested by Fig. 9.42.

First, suppose the penultimate node is v. Then the length of the path from s to v to u suggested in Fig. 9.42 is $dist(v)$ plus the label of the arc $v \to u$.

Alternatively, suppose the penultimate node is some other node w. By inductive hypothesis (a), the shortest path from s to w consists only of nodes that were settled prior to v, and therefore, v does not appear on the path. Thus, the length of the shortest special path to u does not change when we add v to the settled nodes.

Now recall that when we settle v, we adjust each $dist(u)$ to be the smaller of

[10] Note that the fact that the labels are nonnegative is vital; if not, the portion of the path from u to v could have a negative distance, resulting in a shorter path to v.

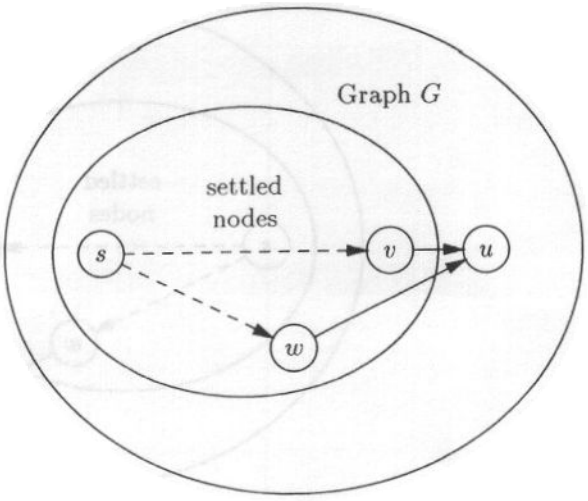


Fig. 9.42. What is the penultimate node on the shortest special path to u?

the old value of $dist(u)$ and $dist(v)$ plus the label of arc $v \to u$. The former covers the case that some w other than v is the penultimate node, and the latter covers the case that v is the penultimate node. Thus, part (b) also holds, and we have completed the inductive step.

Data Structures for Dijkstra's Algorithm

We shall now present an efficient implementation of Dijkstra's algorithm making use of the balanced partially ordered tree structure of Section 5.9.[11] We use two arrays, one called **graph** to represent the graph, and the other called **potNodes** to represent the partially ordered tree. The intent is that to each graph node u there corresponds a partially ordered tree node a that has priority equal to $dist(u)$. However, unlike Section 5.9, we shall organize the partially ordered tree by least priority rather than greatest. (Alternatively, we could take the priority of a to be $-dist(u)$.) Figure 9.43 illustrates the data structure.

We use **NODE** for the type of graph nodes. As usual, we shall name nodes with integers starting at 0. We shall use the type **POTNODE** for the type of nodes in the partially ordered tree. As in Section 5.9, we shall assume that the nodes of the partially ordered tree are numbered starting at 1 for convenience. Thus, both **NODE** and **POTNODE** are synonyms for **int**.

The data type **GRAPH** is defined to be

```
typedef struct {
    float dist;
    LIST successors;
    POTNODE toPOT;
} GRAPH[MAX];
```

[11] Actually, this implementation is only best when the number of arcs is somewhat less than the square of the number of nodes, which is the maximum number of arcs there can be. A simple implementation for the dense case is discussed in the exercises.

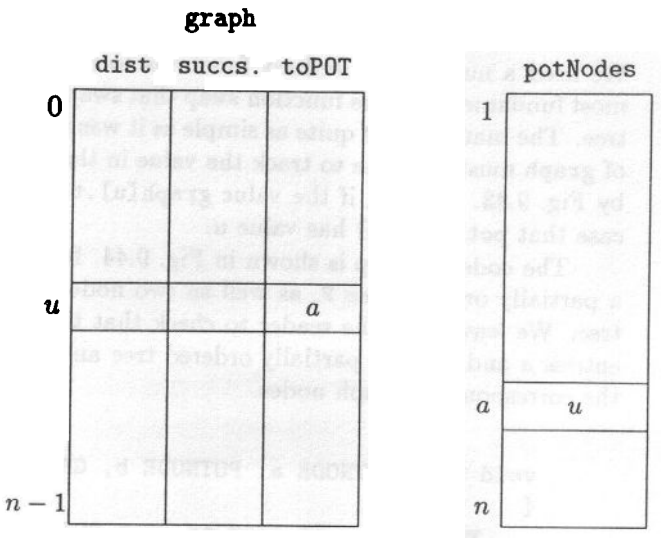


Fig. 9.43. Data structure to represent a graph for Dijkstra's algorithm.

Here, `MAX` is the number of nodes in the graph, and `LIST` is the type of adjacency lists consisting of cells of type `CELL`. Since we need to include labels, which we take to be floating-point numbers, we shall declare as the type `CELL`

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    float nodeLabel;
    LIST next;
};
```

We declare the data type `POT` to be an array of graph nodes

```
typedef NODE POT[MAX+1];
```

We can now define the principal data structures:

```
GRAPH graph;
POT potNodes;
POTNODE last;
```

The array of structures `graph` contains the nodes of the graph, the array `potNodes` contains the nodes of the partially ordered tree, and the variable *last* indicates the current end of the partially ordered tree, which resides in `potNodes[1..last]`.

Intuitively, the structure of the partially ordered tree is represented by the positions in the array `potNodes`, as usual for a partially ordered tree. The elements of this array let us tell the priority of a node by referring back to the graph itself. In particular, we place in `potNodes[a]` the index u of the graph node represented. The `dist` field, `graph[u].dist`, gives the priority of node `a` in the partially ordered tree.

Auxiliary Functions for Dijkstra's Algorithm

We need a number of auxiliary functions to make our implementation work. The most fundamental is the function **swap** that swaps two nodes of the partially ordered tree. The matter is not quite as simple as it was in Section 5.9. Here, the field **toPOT** of **graph** must continue to track the value in the array **potNodes**, as was suggested by Fig. 9.43. That is, if the value **graph[u].toPOT** is a, then it must also be the case that **potNodes[a]** has value u.

The code for **swap** is shown in Fig. 9.44. It takes as arguments a graph **G** and a partially ordered tree **P**, as well as two nodes **a** and **b** of that partially ordered tree. We leave it to the reader to check that the function exchanges the values in entries a and b of the partially ordered tree and also exchanges the **toPOT** fields of the corresponding graph nodes.

```
void swap(POTNODE a, POTNODE b, GRAPH G, POT P)
{
    NODE temp; /* used to swap POT nodes */

    temp = P[b];
    P[b] = P[a];
    P[a] = temp;
    G[P[a]].toPOT = a;
    G[P[b]].toPOT = b;
}
```

Fig. 9.44. Function to swap two nodes of the partially ordered tree.

We shall need to bubble nodes up and down the partially ordered tree, as we did in Section 5.9. The major difference is that here, the value in an element of the array **potNodes** is not the priority. Rather, that value takes us to a node of **graph**, and in the structure for that node we find the field **dist**, which gives us the priority. We therefore need an auxiliary function **priority** that returns **dist** for the appropriate graph node. We shall also assume for this section that smaller priorities rise to the top of the partially ordered tree, rather than larger priorities as in Section 5.9.

Figure 9.45 shows the function **priority** and functions **bubbleUp** and **bubbleDown** that are simple modifications of the functions of the same name in Section 5.9. Each takes a graph **G** and a partially ordered tree **P** as arguments. Function **bubbleDown** also needs an integer **last** that indicates the end of the current partially ordered tree in the array **P**.

Initialization

We shall assume that the adjacency list for each graph node has already been created and that a pointer to the adjacency list for graph node u appears in **graph[u].successors**. We shall also assume that node 0 is the source node. If we take the graph node i to correspond to node $i+1$ of the partially ordered tree, then the array **potNodes** is appropriately initialized as a partially ordered tree. That is, the root of the partially ordered tree represents the source node of the graph, to

```
float priority(POTNODE a, GRAPH G, POT P)
{
    return G[P[a]].dist;
}

void bubbleUp(POTNODE a, GRAPH G, POT P)
{
    if ((a > 1) &&
            (priority(a, G, P) < priority(a/2, G, P))) {
        swap(a, a/2, G, P);
        bubbleUp(a/2, G, P);

}

void bubbleDown(POTNODE a, GRAPH G, POT P, int last)
{
    POTNODE child;

    child = 2*a;
    if (child < last &&
           priority(child+1, G, P) < priority(child, G, P))
        ++child;
    if (child <= last &&
            priority(a, G, P) > priority(child, G, P)) {
        swap(a, child, G, P);
        bubbleDown(child, G, P, last);

}
```

Fig. 9.45. Bubbling nodes up and down the partially ordered tree.

which we give priority 0, and to all other nodes we give priority `INFTY`, our "infinite" defined constant.

As we shall see, on the first round of Dijkstra's algorithm, we select the source node to "settle," which will create the condition we regard as our starting point in the informal introduction, where the source node is settled and `dist[u]` is noninfinite only when there is an arc from the source to u. The initialization function is shown in Fig. 9.46. As with previous functions in this section, `initialize` takes as arguments the graph and the partially ordered tree. It also takes a pointer `pLast` to the integer `last`, so it can initialize it to MAX, the number of nodes in the graph. Recall that `last` will indicate the last position in the array for the partially ordered tree that is currently in use.

Note that the indexes of the partially ordered tree are 1 through MAX, while for the graph, they are 0 through $MAX - 1$. Thus, in lines (3) and (4) of Fig. 9.46, we have to make node i of the graph correspond initially to node $i+1$ of the partially ordered tree.

Implementation of Dijkstra's Algorithm

Figure 9.47 shows the code for Dijkstra's algorithm, using all the functions we

```
    void initialize(GRAPH G, POT P, int *pLast);
    {
        int i; /* we use i as both a graph and a tree node */

(1)     for (i = 0; i < MAX; i++) {
(2)         G[i].dist = INFTY;
(3)         G[i].toPOT = i+1;
(4)         P[i+1] = i;
        }
(5)     G[0].dist = 0;
(6)     (*pLast) = MAX;
    }
```

Fig. 9.46. Initialization for Dijkstra's algorithm

Initializing with an Exception

Notice that at line (2) of Fig 9.46, we set `dist[1]` to `INFTY`, along with all the other distances. Then at line (5), we correct this distance to 0. That is more efficient than testing each value of i to see if it is the exceptional case. True, we could eliminate line (5) if we replaced line (2) by

```
if (i == 0)
    G[i].dist = 0;
else
    G[i].dist = INFTY;
```

but that would not only add to the code, it would increase the running time, since we would have to do n tests and n assignments, instead of $n + 1$ assignments and no tests, as we did in lines (2) and (5) of Fig. 9.46.

have previously written. To execute Dijkstra's algorithm on the graph `graph` with partially ordered tree `potNodes` and with integer `last` to indicate the end of the partially ordered tree, we initialize these variables and then call

```
Dijkstra(graph, potNodes, &last)
```

The function `Dijkstra` works as follows. At line (1) we call `initialize`. The remainder of the code, lines (2) through (13), is a loop, each iteration of which corresponds to one round of Dijkstra's algorithm, where we pick one node v and settle it. The node v picked at line (3) is always the one whose corresponding tree node is at the root of the partially ordered tree. At line (4), we take v out of the partially ordered tree, by swapping it with the current last node of that tree. Line (5) actually removes v by decrementing `last`. Then line (6) restores the partially ordered tree property by calling `bubbleDown` on the node we just placed at the root. In effect, unsettled nodes appear below *last* and settled nodes are at *last* and above.

At line (7) we begin updating distances to reflect the fact that v is now settled. Pointer p is initialized to the beginning of the adjacency list for node v. Then in the loop of lines (8) to (13), we consider each successor u of v. After setting variable

```
    void Dijkstra(GRAPH G, POT P, int *pLast)
    {
        NODE u, v; /* v is the node we select to settle */
        LIST ps; /* ps runs down the list of successors of v;
                u is the successor pointed to by ps */

(1)     initialize(G, P, pLast);
(2)     while ((*pLast) > 1) {
(3)         v = P[1];
(4)         swap(1, *pLast, G, P);
(5)         --(*pLast);
(6)         bubbleDown(1, G, P, *pLast);
(7)         ps = G[v].successors;
(8)         while (ps != NULL) {
(9)             u = ps->nodeName;
(10)            if (G[u].dist > G[v].dist + ps->nodeLabel) {
(11)                G[u].dist = G[v].dist + ps->nodeLabel;
(12)                bubbleUp(G[u].toPOT, G, P);
                }
(13)            ps = ps->next;
            }
        }
    }
```

Fig. 9.47. The main function for Dijkstra's algorithm.

u to one of the successors of v at line (9), we test at line (10) whether the shortest special path to u goes through v. That is the case whenever the old value of $dist(u)$, represented in this data structure by `G[u].dist`, is greater than the sum of $dist(v)$ plus the label of the arc $v \rightarrow u$. If so, then at line (11), we set $dist(u)$ to its new, smaller value, and at line (12) we call `bubbleUp`, so, if necessary, u can rise in the partially ordered tree to reflect its new priority. The loop completes when at line (13) we move p down the adjacency list of v.

Running Time of Dijkstra's Algorithm

As in previous sections, we shall assume that our graph has n nodes and that m is the larger of the number of arcs and the number of nodes. We shall analyze the running time of each of the functions, in the order they were described. First, `swap` clearly takes $O(1)$ time, since it consists only of assignment statements. Likewise, `priority` takes $O(1)$ time.

Function `bubbleUp` is recursive, but its running time is $O(1)$ plus the time of a recursive call on a node that is half the distance to the root. As we argued in Section 5.9, there are at most $\log n$ calls, each taking $O(1)$ time, for a total of $O(\log n)$ time for `bubbleUp`. Similarly, `bubbleDown` takes $O(\log n)$ time.

Function `initialize` takes $O(n)$ time. That is, the loop of lines (1) to (4) is iterated n times, and its body takes $O(1)$ time per iteration. That gives $O(n)$ time for the loop. Lines (5) and (6) each contribute $O(1)$, which we may neglect.

Now let us turn our attention to function `Dijkstra` in Fig. 9.47. Let m_v be the out-degree of node v, or equivalently, the length of v's adjacency list. Begin by

analyzing the inner loop of lines (8) to (13). Each of lines (9) to (13) take $O(1)$ time, except for line (12), the call to **bubbleUp**, which we argued takes $O(\log n)$ time. Thus, the body of the loop takes $O(\log n)$ time. The number of times around the loop equals the length of the adjacency list for v, which we referred to as m_v. Thus the running time of the loop of lines (8) through (13) may be taken as $O(1 + m_v \log n)$; the term 1 covers the case where v has no successors, that is, $m_v = 0$, yet we still do the test of line (8).

Now consider the outer loop of lines (2) through (13). We already accounted for lines (8) to (13). Line (6) takes $O(\log n)$ for a call to **bubbleDown**. The other lines of the body take $O(1)$ each. The body thus takes time $O((1 + m_v) \log n)$.

The outer loop is iterated exactly $n - 1$ times, as **last** ranges from n down to 2. The term 1 in $1 + m_v$ thus contributes $n - 1$, or $O(n)$. However, the m_v term must be summed over each node v, since all nodes (but the last) are chosen once to be v. Thus, the contribution of m_v summed over all iterations of the outer loop is $O(m)$, since $\sum_v m_v \leq m$. We conclude that the outer loop takes time $O(m \log n)$. The additional time for line (1), the call to **initialize**, is only $O(n)$, which we may neglect. Our conclusion is that Dijkstra's algorithm takes time $O(m \log n)$, that is, at most a factor of $\log n$ more than the time taken just to look at the nodes and arcs of the graph.

EXERCISES

9.8.1: Find the shortest distance from Detroit to the other cities, according to the graph of Fig. 9.21 (see the exercises for Section 9.4). If a city is unreachable from Detroit, the minimum distance is "infinity."

9.8.2: Sometimes, we wish to count the number of arcs traversed getting from one node to another. For example, we might wish to minimize the number of transfers needed in a plane or bus trip. If we label each arc 1, then a minimum-distance calculation will count arcs. For the graph in Fig. 9.5 (see the exercises for Section 9.2), find the minimum number of arcs needed to reach each node from node a.

9.8.3: In Fig. 9.48(a) are seven species of hominids and their convenient abbreviations. Certain of these species are known to have preceded others because remains have been found in the same place separated by layers indicating that time had elapsed. The table in Fig. 9.48(b) gives triples (x, y, t) that mean species x has been found in the same place as species y, but x appeared t millions of years before y.

a) Draw a directed graph representing the data of Fig. 9.48, with arcs from the earlier species to the later, labeled by the time difference.
b) Run Dijkstra's algorithm on the graph from (a), with AF as the source, to find the shortest time by which each of the other species could have followed AF.

9.8.4*: The implementation of Dijkstra's algorithm that we gave takes $O(m \log n)$ time, which is less than $O(n^2)$ time, except in the case that the number of arcs is close to n^2, its maximum possible number. If m is large, we can devise another implementation, without a priority queue, where we take $O(n)$ time to select the winner at each round, but only $O(m_u)$ time, that is, time proportional to the number of arcs out of the settled node u, to update *dist*. The result is an $O(n^2)$ time algorithm. Develop the ideas suggested here, and write a C program for this implementation of Dijkstra's algorithm.

Australopithecus Afarensis	AF
Australopithecus Africanus	AA
Homo Habilis	HH
Australopithecus Robustus	AR
Homo Erectus	HE
Australopithecus Boisei	AB
Homo Sapiens	HS

(a) Species and abbreviations.

SPECIES 1	SPECIES 2	TIME
AF	HH	1.0
AF	AA	0.8
HH	HE	1.2
HH	AB	0.5
HH	AR	0.3
AA	AB	0.4
AA	AR	0.6
AB	HS	1.7
HE	HS	0.8

(b) Species 1 precedes species 2 by time.

Fig. 9.48. Relationships between hominid species.

9.8.5**: Dijkstra's algorithm does not always work if there are negative labels on some arcs. Give an example of a graph with some negative labels for which Dijkstra's algorithm gives the wrong answer for some minimum distance.

9.8.6**: Let G be a graph for which we have run Dijkstra's algorithm and settled the nodes in some order. Suppose we add to G an arc $u \to v$ with a weight of 0, to form a new graph G'. Under what conditions will Dijkstra's algorithm run on G' settle the nodes in the same order as for G?

9.8.7*: In this section we took the approach of linking the arrays representing the graph G and the partially ordered tree by storing integers that were indices into the other array. Another approach is to use pointers to array elements. Reimplement Dijkstra's algorithm using pointers instead of integer indices.

9.9 Floyd's Algorithm for Shortest Paths

If we want the minimum distances between all pairs of nodes in a graph with n nodes, with nonnegative labels, we can run Dijkstra's algorithm with each of the n nodes as source. Since one run of Dijkstra's algorithm takes $O(m \log n)$ time, where m is the larger of the number of nodes and number of arcs, finding the minimum distances between all pairs of nodes this way takes $O(mn \log n)$ time. Moreover, if m is close to its maximum, n^2, we can use an $O(n^2)$-time implementation of

Dijkstra's algorithm discussed in Exercise 9.8.4, which when run n times gives us an $O(n^3)$-time algorithm to find the minimum distances between each pair of nodes.

There is another algorithm for finding the minimum distances between all pairs of nodes, called *Floyd's algorithm*. This algorithm takes $O(n^3)$ time, and thus is in principle no better than Dijkstra's algorithm, and worse than Dijkstra's algorithm when the number of arcs is much less than n^2. However, Floyd's algorithm works on an adjacency matrix, rather than adjacency lists, and it is conceptually much simpler than Dijkstra's algorithm.

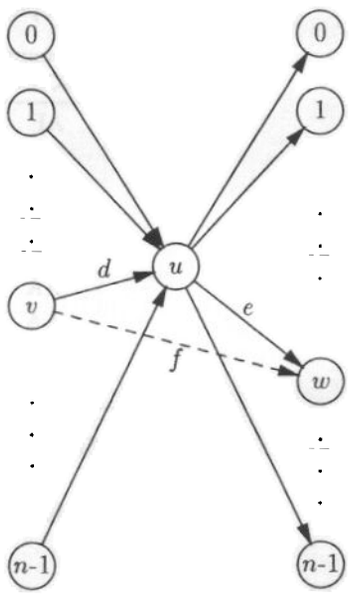


Fig. 9.49. Using node u as a pivot to improve the distances between some pairs of nodes.

Pivot

The essence of Floyd's algorithm is that we consider in turn each node u of the graph as a *pivot*. When u is the pivot, we try to take advantage of u as an intermediate node between all pairs of nodes, as suggested in Fig. 9.49. For each pair of nodes, say v and w, if the sum of the labels of arcs $v \to u$ and $u \to w$, which is $d + e$ in Fig. 9.49, is less than the current label, f, of the arc from v to w, then we replace f by $d + e$.

A fragment of code implementing Floyd's algorithm is shown in Fig. 9.50. As before, we assume nodes are named by integers starting at 0. We use **NODE** as the type of nodes, but we assume this type is integers or an equivalent enumerated type. We assume there is an $n \times n$ array **arc**, such that **arc[v][w]** is the label of the arc $v \to w$ in the given graph. However, on the diagonal we have **arc[v][v]** $= 0$ for all nodes v, even if there is an arc $v \to v$. The reason is that the shortest distance from a node to itself is always 0, and we do not wish to follow any arcs at all. If there is no arc from v to w, then we let **arc[v][w]** be **INFTY**, a special value that is much greater than any other label. There is a similar array **dist** that, at the end, holds the minimum distances; **dist[v][w]** will become the minimum distance from node v to node w.

Lines (1) to (3) initialize **dist** to be **arc**. Lines (4) to (8) form a loop in which

```
        NODE u, v, w;

(1)     for (v = 0; v < MAX; v++)
(2)         for (w = 0; w < MAX; w++)
(3)             dist[v][w] = arc[v][w];
(4)     for (u = 0; u < MAX; u++)
(5)         for (v = 0; v < MAX; v++)
(6)             for (w = 0; w < MAX; w++)
(7)                 if (dist[v][u] + dist[u][w] < dist[v][w])
(8)                     dist[v][w] = dist[v][u] + dist[u][w];
```

Fig. 9.50. Floyd's algorithm.

Warshall's Algorithm

Sometimes, we are only interested in telling whether there exists a path between two nodes, rather than what the minimum distance is. If so, we can use an adjacency matrix where the type of elements is **BOOLEAN** (**int**), with **TRUE** (1) indicating the presence of an arc and **FALSE** (0) its absence. Similarly, the elements of the **dist** matrix are of type **BOOLEAN**, with **TRUE** indicating the existence of a path and **FALSE** indicating that no path between the two nodes in question is known. The only modification we need to make to Floyd's algorithm is to replace lines (7) and (8) of Fig. 9.50 by

```
(7)   if (dist[v][w] == FALSE)
(8)       dist[v][w] = dist[v][u] && dist[u][w];
```

These lines will set **dist[v][w]** to **TRUE**, if it is not already **TRUE**, whenever both **dist[v][u]** and **dist[u][w]** are **TRUE**.

The resulting algorithm, called *Warshall's algorithm*, computes the reflexive and transitive closure of a graph of n nodes in $O(n^3)$ time. That is never better than the $O(nm)$ time that the method of Section 9.7 takes, where we used depth-first search from each node. However, Warshall's algorithm uses an adjacency matrix rather than lists, and if m is near n^2, it may actually be more efficient than multiple depth-first searches because of the simplicity of Warshall's algorithm.

each node u is taken in turn to be the pivot. For each pivot u, in a double loop on v and w, we consider each pair of nodes. Line (7) tests whether it is shorter to go from v to w through u than directly, and if so, line (8) lowers **dist[v][w]** to the sum of the distances from v to u and from u to w.

✦ **Example 9.27.** Let us work with the graph of Fig. 9.10 from Section 9.3, using the numbers 0 through 5 for the nodes; 0 is Laie, 1 is Kaneohe, and so on. Figure 9.51 shows the **arc** matrix, with label **INFTY** for any pair of nodes that do not have a connecting edge. The **arc** matrix is also the initial value of the **dist** matrix.

Note that the graph of Fig. 9.10 is undirected, so the matrix is symmetric; that is, **arc[v][w]** = **arc[v][w]**. If the graph were directed, this symmetry might not

be present, but Floyd's algorithm takes no advantage of symmetry, and thus works for directed or undirected graphs.

	0	1	2	3	4	5
0	0	24	INFTY	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	INFTY
2	INFTY	11	0	13	INFTY	INFTY
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	INFTY	INFTY	12	15	0

Fig. 9.51. The arc matrix, which is the initial value of the `dist` matrix.

The first pivot is $u = 0$. Since the sum of `INFTY` and anything is `INFTY`, the only pair of nodes v and w, neither of which is u, for which `dist[v][u] + dist[u][w]` is less than `INFTY` is $v = 1$ and $w = 5$, or vice versa.[12] Since `dist[1][5]` is `INFTY` at this time, we replace `dist[1][5]` by the sum of `dist[1][0] + dist[0][5]` which is 52. Similarly, we replace `dist[5][1]` by 52. No other distances can be improved with pivot 0, which leaves the `dist` matrix of Fig. 9.52.

	0	1	2	3	4	5
0	0	24	INFTY	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	52
2	INFTY	11	0	13	INFTY	INFTY
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	INFTY	12	15	0

Fig. 9.52. The matrix `dist` after using 0 as the pivot.

Now we make node 1 the pivot. In the current `dist`, shown in Fig. 9.52, node 1 has noninfinite connections to 0 (distance 24), 2 (distance 11), and 5 (distance 52). We can combine these edges to reduce the distance between nodes 0 and 2 from `INFTY` to $24+11 = 35$. Also, the distance between 2 and 5 is reduced to $11+52 = 63$. Note that 63 is the distance along the path from Honolulu, to Kaneohe, to Laie, to Wahiawa, not the shortest way to get to Wahiawa, but the shortest way that only goes through nodes that have been the pivot so far. Eventually, we shall find the shorter route through Pearl City. The current `dist` matrix is shown in Fig. 9.53.

Now we make 2 be the pivot. Node 2 currently has noninfinite connections to 0 (distance 35), 1 (distance 11), 3 (distance 13), and 5 (distance 63). Among these nodes, the distance between 0 and 3 can be improved to $35 + 13 = 48$, and the

[12] If one of v and w is the u, it is easy to see `dist[v][w]` can never be improved by going through u. Thus, we can ignore pairs of the form (v, u) or (u, w) when searching for pairs whose distance is improved by going through the pivot u.

	0	1	2	3	4	5
0	0	24	35	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	52
2	35	11	0	13	INFTY	63
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	63	12	15	0

Fig. 9.53. The matrix `dist` after using 1 as the pivot.

	0	1	2	3	4	5
0	0	24	35	48	INFTY	28
1	24	0	11	24	INFTY	52
2	35	11	0	13	INFTY	63
3	48	24	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	63	12	15	0

Fig. 9.54. The matrix `dist` after using 2 as the pivot.

distance between 1 and 3 can be improved to $11 + 13 = 24$. Thus, the current `dist` matrix is shown in Fig. 9.54.

Next, node 3 becomes the pivot. Figure 9.55 shows the current best distance between 3 and each of the other nodes.[13] By traveling through node 3, we can make the following improvements in distances.

1. Between 1 and 5, the distance is reduced to 36.
2. Between 2 and 5, the distance is reduced to 25.
3. Between 0 and 4, the distance is reduced to 68.
4. Between 1 and 4, the distance is reduced to 44.
5. Between 2 and 4, the distance is reduced to 33.

The current `dist` matrix is shown in Fig. 9.56.

The use of 4 as a pivot does not improve any distances. When 5 is the pivot, we can improve the distance between 0 and 3, since in Fig. 9.56,

```
dist[0][5] + dist[5][3] = 40
```

[13] The reader should compare Fig. 9.55 with Fig. 9.49. The latter shows how to use a pivot node in the general case of a directed graph, where the arcs in and out of the pivot may have different labels. Fig. 9.55 takes advantage of the symmetry in the example graph, letting us use edges between node 3 and the other nodes to represent both arcs into node 3, as on the left of Fig. 9.49, and arcs out of 3, as on the right of Fig. 9.49.

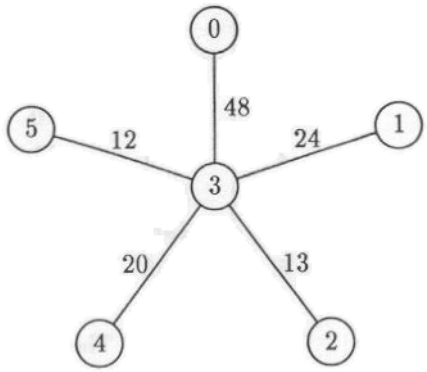


Fig. 9.55. Current best distances to node 4.

	0	1	2	3	4	5
0	0	24	35	48	68	28
1	24	0	11	24	44	36
2	35	11	0	13	33	25
3	48	24	13	0	20	12
4	68	44	33	20	0	15
5	28	36	25	12	15	0

Fig. 9.56. The matrix `dist` after using 3 as the pivot.

	0	1	2	3	4	5
0	0	24	35	40	43	28
1	24	0	11	24	44	36
2	35	11	0	13	33	25
3	40	24	13	0	20	12
4	43	44	33	20	0	15
5	28	36	25	12	15	0

Fig. 9.57. The final `dist` matrix.

which is less than `dist[0][3]`, or 48. In terms of cities, that corresponds to discovering that it is shorter to go from Laie to Pearl City via Wahiawa than via Kaneohe and Honolulu. Similarly, we can improve the distance between 0 and 4 to 43, from 68. The final `dist` matrix is shown in Fig. 9.57. ✦

Why Floyd's Algorithm Works

As we have seen, at any stage during Floyd's algorithm the distance from node v to node w will be the distance of the shortest of those paths that go through only nodes that have been the pivot. Eventually, all nodes get to be the pivot, and `dist[v][w]` holds the minimum distance of all possible paths.

k-path

We define a *k-path* from a node v to a node w to be a path from v to w such that no intermediate node is numbered higher than k. Note that there is no constraint that v or w be k or less.

An important special case is when $k = -1$. Since nodes are assumed numbered starting at 0, a (-1)-path can have no intermediate nodes at all. It can only be either an arc or a single node that is both the beginning and end of a path of length 0.

Figure 9.58 suggests what a k-path looks like, although the end points, v and w, can be above or below k. In that figure, the height of the line represents the numbers of the nodes along the path from v to w.

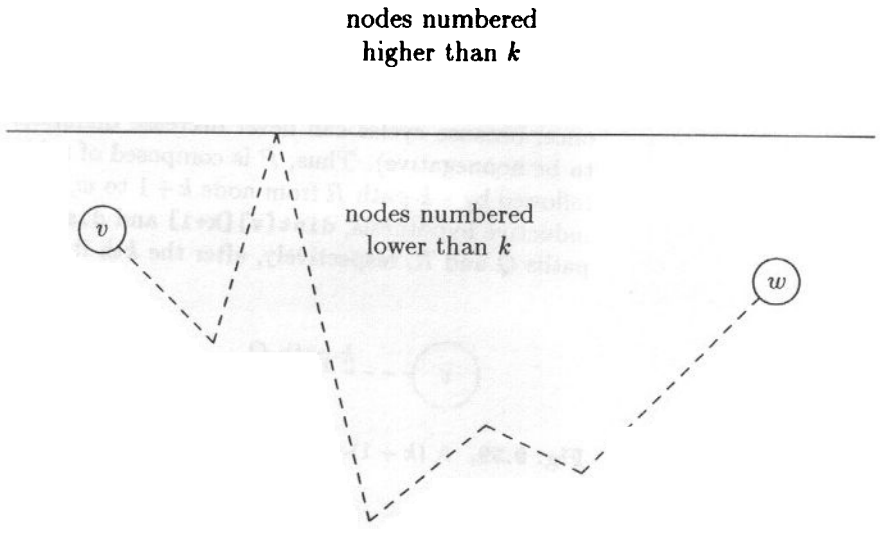


Fig. 9.58. A k-path cannot have nodes higher than k, except (possibly) at the ends.

✦ **Example 9.28.** In Fig. 9.10, the path 0, 1, 2, 3 is a 2-path. The intermediate nodes, 1 and 2, are each 2 or less. This path is also a 3-path, a 4-path, and a 5-path. It is not a 1-path, however, because the intermediate node 2 is greater than 1. Similarly, it is not a 0-path or a (-1)-path. ✦

As we assume nodes are numbered 0 to $n-1$, a (-1)-path cannot have any intermediate nodes at all, and thus must be an arc or a single node. An $(n-1)$-path is any path at all, since there can be no intermediate node numbered higher than $n-1$ in any path of a graph with nodes numbered 0 through $n-1$. We shall prove by induction on k the statement

STATEMENT $S(k)$: If labels of arcs are nonnegative, then just before we set u to $k+1$ in the loop of lines (4) to (8) of Fig. 9.50, `dist[v][w]` is the length of the shortest k-path from v to w, or `INFTY` if there is no such path.

BASIS. The basis is $k = -1$. We set u to 0 just before we execute the body of the loop for the first time. We have just initialized `dist` to be `arc` in lines (1) to (3). Since the arcs and the paths consisting of a nod by itself are the only (-1)-paths, the basis holds.

INDUCTION. Assume $S(k)$, and consider what happens to `dist[v][w]` during the iteration of the loop with $u = k + 1$. Suppose is a shortest $(k+1)$-path from v to w. There are two cases, depending on wheth P goes through node $k+1$.

1. If P is a k-path, that is, P does not actua y go through node $k+1$, then by the inductive hypothesis, `dist[v][w]` alrea y equals the length of P after the kth iteration. We cannot change `dist[u]` `v]` during the round with $k+1$ as pivot, because there are no shorter $(k+1)$-paths.

2. If P is a $(k+1)$-path, we can assume tha P only goes through node $k+1$ once, because cycles can never decrease di tances (recall we require all labels to be nonnegative). Thus, P is composed o a k-path Q from v to node $k+1$, followed by a k-path R from node $k+1$ to , as suggested in Fig. 9.59. By the inductive hypothesis, `dist[v][k+1]` and `st[k+1][w]` will be the lengths of paths Q and R, respectively, after the kth teration.

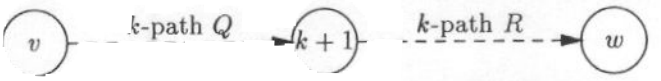


Fig. 9.59. A $(k+1)$-path P can be broken into two k-paths, Q followed by R.

Let us begin by observing that `dist[v][` `+1]` and `dist[k+1][w]` cannot be changed in the $(k+1)$st iteration. The reason i that all arc labels are nonnegative, and so all lengths of paths are nonnegative; t us the test of line (7) in Fig. 9.50 must fail when u (i.e., node $k+1$) is one of v o w.

Thus, when we apply the test of line (7) fo arbitrary v and w, with $u = k+1$, the values of `dist[v][k+1]` and `dist[k+1][w]` have not changed since the end of the kth iteration. That is to say, the test of ine (7) compares the length of the shortest k-path, with the sum of the lengths of he shortest k-paths from v to $k+1$ and from $k+1$ to w. In case (1), where path P does not go through $k+1$, the former will be the shorter, and in case (2), w ere P does go through $k+1$, the latter will be the sum of the lengths of the patl s Q and R in Fig. 9.59, and will be the shorter.

We conclude that the $(k+1)$st iteration s ts `dist[v][w]` to the length of the shortest $(k+1)$-path, for all nodes v and w. That is the statement $S(k+1)$, and so we conclude the induction.

To finish our proof, we let $k = n - 1$. Tha is, we know that after finishing all n iterations, `dist[v][w]` is the minimum dist nce of any $(n-1)$-path from v to w. But since any path is an $(n-1)$-path, we ave shown that `dist[v][w]` is the minimum distance along any path from v to w

EXERCISES

9.9.1: Assuming all arcs in Fig. 9.5 (see the exercises for Section 9.2) have label 1, use Floyd's algorithm to find the length of the shortest path between each pair of nodes. Show the distance matrix after pivoting with each node.

9.9.2: Apply Warshall's algorithm to the graph of Fig. 9.5 to compute its reflexive and transitive closure. Show the reachability matrix after pivoting with each node.

9.9.3: Use Floyd's algorithm to find the shortest distances between each pair of cities in the graph of Michigan in Fig. 9.21 (see the exercises for Section 9.4).

9.9.4: Use Floyd's algorithm to find the shortest possible time between each of the hominid species in Fig. 9.48 (see the exercises for Section 9.8).

9.9.5: Sometimes we want to consider only paths of one or more arcs, and exclude single nodes as paths. How can we modify the initialization of the `arc` matrix so that only paths of length 1 or more will be considered when finding the shortest path from a node to itself?

9.9.6*: Find all the acyclic 2-paths in Fig. 9.10.

9.9.7*: Why does Floyd's algorithm not work when there are both positive and negative costs on the arcs?

9.9.8**: Give an algorithm to find the longest acyclic path between two given nodes.

9.9.8**: Suppose we run Floyd's algorithm on a graph G. Then, we lower the label of the arc $u \rightarrow v$ to 0, to construct the new graph G'. For what pairs of nodes s and t will `dist[s][t]` be the same at each round when Floyd's algorithm is applied to G and G'?

9.10 An Introduction to Graph Theory

Graph theory is the branch of mathematics concerned with properties of graphs. In the previous sections, we have presented the basic definitions of graph theory, along with some fundamental algorithms that computer scientists have developed to calculate key properties of graphs efficiently. We have seen algorithms for computing shortest paths, spanning trees, and depth-first-search trees. In this section, we shall present a few more important concepts from graph theory.

Complete Graphs

An undirected graph that has an edge between every pair of distinct nodes is called a *complete* graph. The complete graph with n nodes is called K_n. Figure 9.60 shows the complete graphs K_1 through K_4.

The number of edges in K_n is $n(n-1)/2$, or $\binom{n}{2}$. To see why, consider an edge $\{u, v\}$ of K_n. For u we can pick any of the n nodes; for v we can pick any of the remaining $n-1$ nodes. The total number of choices is therefore $n(n-1)$. However, we count each edge twice that way, once as $\{u, v\}$ and a second time as $\{v, u\}$, so that we must divide the total number of choices by 2 to get the correct number of edges.

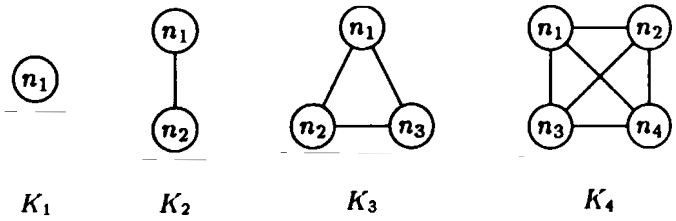


Fig. 9.60. The first four complete graphs.

Complete directed graph

There is also a notion of a complete directed graph. This graph has an arc from every node to every other node, including itself. A complete directed graph with n nodes has n^2 arcs. Figure 9.61 shows the complete directed graph with 3 nodes and 9 arcs.

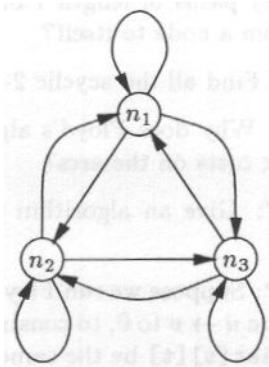


Fig. 9.61. The complete directed graph with three nodes.

Planar Graphs

An undirected graph is said to be *planar* if it is possible to place its nodes on a plane and then draw its edges as continuous lines so that no two edges cross.

✦ **Example 9.29.** The graph K_4 was drawn in Fig. 9.60 in such a way that its two diagonal edges crossed. However, K_4 is a planar graph, as we can see by the drawing in Fig. 9.62. There, by redrawing one of the diagonals on the outside, we avoid having any two edges cross. We say that Fig. 9.62 is a *plane presentation* of the graph K_4, while the drawing in Fig. 9.60 is a nonplane presentation of K_4. Note that it is permissible to have edges that are not straight lines in a plane presentation. ✦

Plane presentation

Nonplanar graph

In Figure 9.63 we see what are in a sense the two simplest *nonplanar* graphs, that is, graphs that do not have any plane presentation. One is K_5, the complete graph with five nodes. The other is sometimes called $K_{3,3}$; it is formed by taking two groups of three nodes and connecting each node of one group to each node of the other group, but not to nodes of the same group. The reader should try to

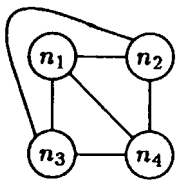
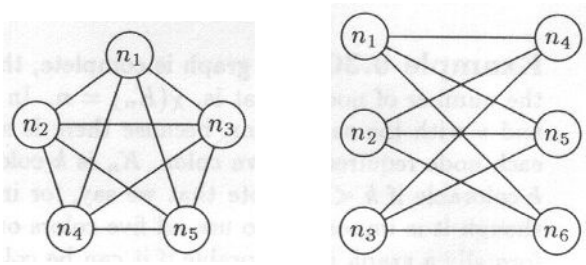


Fig. 9.62. A plane presentation of K_4.



K_5 $K_{3,3}$

Fig. 9.63. The two simplest nonplanar graphs.

redraw each of these graphs so that no two edges cross, just to get a feel for why they are not planar.

Kuratowski's theorem

A famous theorem by Kuratowski states every nonplanar graph contains a "copy" of at least one of these two graphs. We must be a little careful in interpreting the notion of a copy, however, since to see a copy of K_5 or $K_{3,3}$ in an arbitrary nonplanar graph G, we may have to associate some edges in the graphs of Fig. 9.63 with paths in the graph G.

Applications of Planarity

Planarity has considerable importance in computer science. For example, many graphs or similar diagrams need to be presented on a computer screen or on paper. For clarity, it is desirable to make a plane presentation of the graph, or if the graph is not planar, to make as few crossings of edges as possible.

The reader may observe that in Chapter 13 we draw some fairly complex diagrams of circuits, which are really graphs whose nodes are gates and junction points of wires, and whose edges are the wires. Since these circuits are not planar in general, we had to adopt a convention in which wires were allowed to cross without connecting, and a dot signals a connection of wires.

A related application concerns the design of integrated circuits. Integrated circuits, or "chips," embody logical circuits such as those discussed in Chapter 13. They do not require that the logical circuit be inscribed in a plane presentation, but there is a similar limitation that allows us to assign edges to several "levels," often three or four levels. On one level, the graph of the circuit must have a plane presentation; edges are not allowed to cross. However, edges on different levels may cross.

Graph Coloring

Chromatic number

The problem of *graph coloring* for a graph G is to assign a "color" to each node so that no two nodes that are connected by an edge are assigned the same color. We may then ask how many distinct colors are required to *color* a graph in this sense. The minimum number of colors needed for a graph G is called the *chromatic number* of G, often denoted $\chi(G)$. A graph that can be colored with no more than k colors is called *k-colorable*.

k-colorability

✦ **Example 9.30.** If a graph is complete, then its chromatic number is equal to the number of nodes; that is, $\chi(K_n) = n$. In proof, we cannot color two nodes u and v with the same color, because there is surely an edge between them. Thus, each node requires its own color. K_n is k-colorable for each $k \geq n$, but K_n is not k-colorable if $k < n$. Note that we say, for instance, that K_4 is 5-colorable, even though it is impossible to use all five colors on the four-node graph K_4. However, formally a graph is k-colorable if it can be colored with k or fewer colors, not only if it is colorable with exactly k colors.

As another example, the graph $K_{3,3}$ shown in Fig. 9.63 has chromatic number 2. For example, we can color the three nodes in the group on the left *red* and color the three nodes on the right *blue*. Then all edges go between a *red* and a *blue* node.

Bipartite graph

$K_{3,3}$ is an example of a *bipartite graph*, which is another name for a graph that can be colored with two colors. All such graphs can have their nodes divided into two groups such that no edge runs between members of the same group.

As a final example, the chromatic number for the six-node graph of Fig. 9.64 is 4. To see why, note that the node in the center cannot have the same color as any other node, since it is connected to all. Thus, we reserve a color for it, say, *red*. We need at least two other colors for the ring of nodes, since neighbors around the ring cannot get the same color. However, if we try alternating colors — say, *blue* and *green* — as we did in Fig. 9.64, then we run into a problem that the fifth node has both *blue* and *green* neighbors, and therefore needs a fourth color, *yellow*, in our example. ✦

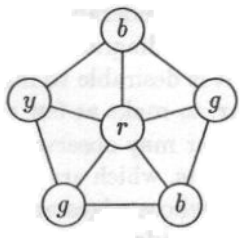


Fig. 9.64. A graph with chromatic number 4.

Applications of Graph Coloring

Finding a good graph coloring is another problem that has many uses in computer science. For example, in our introduction to the first chapter, we considered assigning courses to time slots so that no pair of courses in the same time slot had a student taking both courses. The motivation was to schedule final exams so that no student had to take two exams at the same time. We drew a graph whose

nodes were the courses, with an edge between two courses if they had a student in common.

The question of how many time slots we need in which to schedule exams can thus be posed as the question of what is the chromatic number of this graph. All nodes of the same color can be scheduled at the same time since they have no edges between any two of them. Conversely, if we have a schedule that does not cause conflicts for any student, then we can color all the courses scheduled at the same time with the same color, and thus produce a graph coloring with as many colors as there are exam periods.

In Chapter 1 we discussed a heuristic based on finding maximal independent sets to schedule the exams. That is a reasonable heuristic for finding a good coloring of a graph as well. One might expect that one could try all possible colorings for a graph as small as the five-node graph in Fig. 1.1, and indeed that is true. However, the number of possible colorings of a graph grows exponentially with the number of nodes, and it is not feasible to consider all possible colorings for significantly larger graphs, in our search for the least possible number of colors.

Cliques

k-clique

Clique number

A *clique* in an undirected graph G is a set of nodes such that there is in G an edge between every pair of nodes in the set. A clique of k nodes is called a *k-clique*. The size of the largest clique in a graph is called the *clique number* of that graph.

✦ **Example 9.31.** As a simple example, every complete graph K_n is a clique consisting of all n nodes. In fact, K_n has a k-clique for all $k \leq n$, but no k-clique if $k > n$.

The graph of Fig. 9.64 has cliques of size three, but no greater. The 3-cliques are each shown as triangles. There cannot be a 4-clique in this graph, because it would have to include some of the nodes in the ring. Each ring node is connected to only three other nodes, so the 4-clique would have to include some node v on the ring, its neighbors on the ring, and the central node. However, the neighbors of v on the ring do not have an edge between them, so we do not have a 4-clique. ✦

As an example application of cliques, suppose we represented conflicts among courses not as in Fig. 1.1, but rather by putting an edge between two nodes if they *did not have* a student enrolled in both courses. Thus, two courses connected by an edge could have their exams scheduled at the same time. We could then look for

Maximal clique

maximal cliques, that is, cliques that were not subsets of larger cliques, and schedule the exams for a maximal clique of courses at the same period.

EXERCISES

9.10.1: For the graph of Fig. 9.4,

a) What is the chromatic number?
b) What is the clique number?
c) Give an example of one largest clique.

9.10.2: What are the chromatic numbers of the undirected versions of the graphs shown in (a) Fig. 9.5 and (b) Fig. 9.26? (Treat arcs as edges.)

9.10.3: Figure 9.5 is not presented in a plane manner. Is the graph planar? That is, can you redraw it so there are no crossing edges?

9.10.4*: Three quantities associated with an undirected graph are its degree (maximum number of neighbors of any node), its chromatic number, and its clique number. Derive inequalities that must hold between these quantities. Explain why they must hold.

9.10.5**: Design an algorithm that will take any graph of n nodes, with m the larger of the number of nodes and edges, and in $O(m)$ time will tell whether the graph is bipartite (2-colorable).

9.10.6*: We can generalize the graph of Fig. 9.64 to have a central node and k nodes in a ring, each node connected only to its neighbors around the ring and to the central node. As a function of k, what is the chromatic number of this graph?

9.10.7*: What can you say about the chromatic number of unordered, unrooted trees (as discussed in Section 9.5)?

9.10.8**: Let $K_{i,j}$ be the graph formed by taking a group of i nodes and a group of j nodes and placing an edge from every member of one group to every member of the other group. We observed that if $i = j = 3$, then the resulting graph is not planar. For what values of i and j is the graph $K_{i,j}$ planar?

9.11 Summary of Chapter 9

The table of Fig. 9.65 summarizes the various problems we have addressed in this chapter, the algorithms for solving them, and the running time of the algorithms. In this table, n is the number of nodes in the graph and m is the larger of the number of nodes and the number of arcs/edges. Unless otherwise noted, we assume graphs are represented by adjacency lists.

In addition, we have introduced the reader to most of the key concepts of graph theory. These include

- Paths and shortest paths
- Spanning trees
- Depth-first search trees and forests
- Graph coloring and the chromatic number
- Cliques and clique numbers
- Planar graphs.

PROBLEM	ALGORITHM(S)	RUNNING TIME
Minimal spanning tree	Kruskal's	$O(m \log n)$
Detecting cycles	Depth-first search	$O(m)$
Topological order	Depth-first search	$O(m)$
Single-source reachability	Depth-first search	$O(m)$
Connected components	Depth-first search	$O(m)$
Transitive closure	n depth-first searches	$O(mn)$
Single-source shortest path	Dijkstra's with POT implementation	$O(m \log n)$
	Dijkstra's with implementation of Exercise 9.8.4	$O(n^2)$
All-pairs shortest path	n uses of Dijkstra with POT implementation	$O(mn \log n)$
	n uses of Dijkstra with implementation of Exercise 9.8.4	$O(n^3)$
	Floyd's, with adjacency matrix	$O(n^3)$

Fig. 9.65. A summary of graph algorithms.

9.12 Bibliographic Notes for Chapter 9

For additional material on graph algorithms, see Aho, Hopcroft, and Ullman [1974, 1983]. Depth-first search was first used to create efficient graph algorithms by Hopcroft and Tarjan [1973]. Dijkstra's algorithm is from Dijkstra [1959], Floyd's algorithm from Floyd [1962], Kruskal's algorithm from Kruskal [1956], and Warshall's algorithm from Warshall [1962].

Berge [1962] covers the mathematical theory of graphs. Lawler [1976], Papadimitriou and Steiglitz [1982], and Tarjan [1983] present advanced graph optimization techniques.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Berge, C. [1962]. *The Theory of Graphs and its Applications*, Wiley, New York.

Dijkstra, E. W. [1959]. "A note on two problems in connexion with graphs," *Numberische Mathematik* **1**, pp. 269–271.

Floyd, R. W. [1962]. "Algorithm 97: shortest path," *Comm. ACM* **5**:6, pp. 345.

Hopcroft, J. E., and R. E. Tarjan [1973]. "Efficient algorithms for graph manipulation," *Comm. ACM* **16**:6, pp. 372-378.

Kruskal, J. B., Jr. [1956]. "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. AMS* **7**:1, pp. 48–50.

Lawler, E. [1976]. *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

Papadimitriou, C. H., and K. Steiglitz [1982]. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey.

Tarjan, R. E. [1983]. *Data Structures and Network Algorithms*, SIAM, Philadelphia.

Warshall, S. [1962]. "A theorem on Boolean matrices," *J. ACM* 9:1, pp. 11-12.